A Schema for Constraint Relaxation with Instantiations for Partial Constraint Satisfaction and Schedule Optimization

by

John Christopher Beck

A thesis submitted in conformity with the requirements

for the degree of Master of Science

Graduate Department of Computer Science

University of Toronto

Abstract

A Schema for Constraint Relaxation with Instantiations for Partial

Constraint Satisfaction and Schedule Optimization

John Christopher Beck

Master of Science

Department of Computer Science

University of Toronto

1994

We investigate constraint relaxation within a general constraint model. We claim that a key to relaxation is recognition that a constraint can be modified in a variety of ways and that each modification potentially carries a different impact for both the quality of the solution and the problem solving process. Our primary motivation is the application of constraint relaxation as a technique for coordination of multiple agents in a shared environment.

We propose a schema for constraint relaxation that is based on the propagation of information through a constraint graph. The schema isolates five heuristic decision points where techniques of varying complexities can be specified. Three algorithms within the schema are declared and shown to perform well on Partial Constraint Satisfaction Problems (PCSPs). Three additional algorithms are defined and used in the estimation of the impact of scheduling decisions in a medium size job shop scheduling problems. Difficulties with the calculation of actual impact data prevents comparison among the algorithms. The algorithms represent an advance by allowing propagation over all types of constraints and the ability to integrate heuristic decision making.

Acknowledgments

I would like to thank all who made this work possible. In particular, I would like to express gratitude to the following people:

> To my parents, John and Anne, for love and support and for teaching me to think.
>
> To my supervisor, Mark Fox, for guidance and encouragement, as well as for unbelievably insightful comments.
>
> To my second reader, Hector Levesque, for his useful comments
>
> To Eugene Davis, Steve Green, John Chappel, Heather Hinton, Yaska Sankar, and Michael Gruninger for wading through early drafts and magnanimously sacrificing both hours and red pens. I would especially like to thank Gene for many discussions about the research from its genesis to its apocalypse.
>
> To Maureen Whelton, Monica Burchill, Helen Tubrett, Frank Beck, and Nuala Beck for continually reminding me that there is a life outside of the lab and for keeping me alive to experience it.
>
> To Angela Glover, who demonstrated that the light at the end of the tunnel was not an oncoming train.
>
> And finally to Nancy Bowes, who asked me to put her name here.

I would also like to acknowledge the financial support of the Natural Sciences and Engineering Research Council and Digital Equipment Corporation.

Contents

Abstract iii

Acknowledgments v

Contents vii

List of Figures xi

List of Tables xiii

Chapter 1 Introduction 1
- 1.1 Constraint Relaxation Overview 1
- 1.2 Motivation 2
  - 1.2.1 Coordination of Multiple Agents 2
  - 1.2.2 Guiding Search with Relaxation 5
  - 1.2.3 Scheduling 6
  - 1.2.4 Constraint-Directed Reasoning 7
- 1.3 Constraint Satisfaction Problems (CSPs) 7
  - 1.3.1 The CSP Model 7
  - 1.3.2 CSP Algorithms 8
    - 1.3.2.1 Retrospective Techniques 8
    - 1.3.2.2 Prospective Techniques 10
    - 1.3.2.3 Variable and Value Ordering 10
    - 1.3.2.4 Texture Measurements 11
- 1.4 Constraint Relaxation 11
  - 1.4.1 Relaxation as Repair and Optimization 12
  - 1.4.2 Relaxation and Selective Non-Satisfaction of Constraints 12
- 1.5 Contributions of this Work 13
- 1.6 Plan of Dissertation 13

Chapter 2 Related Work 15
- 2.1 Probabilistic Labeling in Machine Vision 15
- 2.2 Non-Satisfaction of Constraints By Weight 16
- 2.3 Extending CSP Solving Methods 17
- 2.4 Constraint Optimization 19
  - 2.4.1 Optimization in Operations Research 20
  - 2.4.2 Optimization in Artificial Intelligence 21
- 2.5 Scheduling 22
  - 2.5.1 Constructive Scheduling 22
  - 2.5.2 Repair-based Scheduling 23

2.5.3 Fuzzy Scheduling 23
2.5.4 Estimating the Impact of a Scheduling Decision 24
2.6 Discussion 26
2.7 Summary 27

Chapter 3 A Schema for Constraint Relaxation 29
3.1 Foundational Concepts 29
3.1.1 Constraint Model 29
3.1.2 Search With Relaxation 30
3.2 Relaxation-Based Search 30
3.2.1 Propagation 31
3.2.2 An Extended Example 32
3.2.3 Comments on the Example 36
3.3 The Relaxation Schema 36
3.4 Basic Propagation 38
3.4.1 Value and Cost Propagation 38
3.4.2 Relaxation Propagation 39
3.4.3 Termination of Propagation 40
3.5 Relaxation in Non-Trivial Graphs 41
3.6 Summary 42

Chapter 4 Cache-Based Constraint Relaxation 43
4.1 Difficulties With Cycles 43
4.1.1 Counting Cycle Costs 43
4.1.2 Relaxation Information 46
4.1.3 Complexity 48
4.2 Using Information Caches to Limit Search 48
4.2.1 Using Stored Costs 49
4.2.2 Bounding the Search 49
4.2.3 Enhancements 50
4.3 Cache-Based Relaxation Algorithms 51
4.3.1 Finding an Initial Solution 52
4.3.2 Single Min-Value Relaxation 52
4.3.3 Multiple Min-Value Relaxation 53
4.3.4 MinConflicts-like Relaxation 54
4.3.5 Complexity 54
4.4 Experiments with PCSPs 55
4.4.1 Partial Constraint Satisfaction Revisited 55
4.4.2 Partial Extended Forward Checking 55
4.5 Experiments 56
4.5.1 Evaluation Criteria 57
4.5.2 Problem Description 57
4.6 Results 58
4.6.1 Preliminary Experiments 58
4.6.2 Consistency Checks 59
4.6.3 Solution Cost 60
4.7 Discussion 61
4.7.1 Non-Uniform Relaxation Costs 61
4.7.1.1 Consistency Checks 61

4.7.1.2 Solution Costs 62
4.7.2 Increasing the Number of Variables 62
4.7.3 Estimation of Global Cost vs. Local Cost 63
4.7.4 The MC Algorithm 64
4.8 Summary 65

Chapter 5 Relaxation-Based Estimation of the Impact of Scheduling Decisions 67
5.1 Estimating the Impact of a Scheduling Decision Revisited 67
5.2 Job Shop Scheduling 68
5.2.1 A Constraint Model for Job Shop Scheduling 68
5.2.2 Temporal Variables and Constraints 69
5.2.3 Resource Variables and Constraints 70
5.2.4 Example 71
5.2.5 Constraint Relaxation in Scheduling 72
5.2.6 Extending the Example 73
5.3 A General Relaxation-Based Algorithm for Cost Estimation 74
5.3.1 The General Estimation Algorithm 74
5.3.2 Finding a Subgraph 76
5.3.3 The Constructive Search Algorithm 77
5.3.3.1 Value and Control Propagation 77
5.3.3.2 Backtracking 78
5.3.3.3 The Relaxation Schema 78
5.3.4 Calculating the Cost of a Subgraph Solution 78
5.3.5 Partial Solutions vs. Solutions to the Subgraph 79
5.3.6 Complexity 80
5.4 The Cost Estimation Algorithms 80
5.4.1 The Depth Algorithm 80
5.4.2 The TemporalOnly Algorithm 80
5.4.3 The TopCost Algorithm 81
5.4.4 The Contention/Reliance Algorithm 81
5.4.5 Expectations 82
5.5 Experiments 82
5.5.1 Experimental Problems 83
5.5.2 Cost Model 83
5.5.3 Evaluation Criteria 85
5.5.4 The Source Activity 86
5.5.4.1 Selecting the Source Activity 86
5.5.4.2 Selecting Candidate Values for the Source Activity 87
5.5.5 The Depth Algorithm 87
5.5.6 Parameter Settings for the Texture-Based Algorithms 89
5.5.6.1 Subgraph Size 89
5.5.6.2 NumberOfSets and SetSize 89
5.5.6.3 Summary of Algorithm Variations 90
5.6 Results 91
5.6.1 Raw Data 91
5.6.2 Texture Measurements 93
5.6.3 Subgraph Size 95
5.6.4 Parameter Settings 97
5.6.5 Overall 98

5.7 Discussion 100
5.7.1 The Depth Algorithm 100
5.7.2 TopCost vs. Baseline 100
5.7.3 Poor Performance of Contention/Reliance 101
5.7.4 TopCost4 vs. TopCost10 101
5.7.5 Reassessing the Estimations 102
5.8 Conclusion 102
5.8.1 Building Practical Algorithms 102
5.9 Summary 103

Chapter 6 Coordination of Multiple Agents 105
6.1 A Theory of Coordination 105
6.2 A Mediated Approach to Coordination 106
6.2.1 The Role of the Mediator 106
6.2.2 Why Mediation? 106
6.3 Supply Chain Management Revisited 107
6.3.1 An Example 107
6.4 Conclusion 110

Chapter 7 Concluding Remarks 113
7.1 Contributions 113
7.1.1 A Generalized Constraint Model 113
7.1.2 A Schema for Relaxation Algorithms 114
7.1.3 Three Relaxation Algorithms for PCSPs 114
7.1.4 Application of the Relaxation Schema to Schedule Optimization 115
7.2 Future Work 115
7.3 Summary 116

Bibliography 117

List of Figures

Figure 1.	The Material Flow in the Supply Chain of a Manufacturing Enterprise	3
Figure 2.	Logistics Level Scheduling of a Customer Order	4
Figure 3.	A Simple Activity/Resource Constraint Graph	5
Figure 4.	The Forward Step of the PEFC3 Algorithm [Freuder 92]	18
Figure 5.	A Simple Constraint Graph	32
Figure 6.	Snapshot of the Graph After Step 6	33
Figure 7.	Snapshot of the Graph After Step 14	34
Figure 8.	The Graph after All Value/Cost Propagation	35
Figure 9.	The Graph of the Final Solution for $x_1 = 2$	35
Figure 10.	Pseudocode for Relaxation Propagation	39
Figure 11.	Redefinition of the PostCost Function to account for Graph Cycles	41
Figure 12.	A Constraint Graph with a Cycle	44
Figure 13.	A Close-up of the Constraint Graph in Figure 12	44
Figure 14.	A Constraint Graph with Cycles	45
Figure 15.	A Close-Up of a Subgraph from the Graph in Figure 14	45
Figure 16.	Overwriting the Cost Cache at a Constraint	47
Figure 17.	A Graph Corresponding to the Worst-Case Space Complexity of the Context Mechanism	48
Figure 18.	Filtering the Cost Bound at a Variable	49
Figure 19.	Filtering the Cost Bound at a Constraint	50
Figure 20.	Pseudocode for the Main Loop of the Cache-Based Relaxation Algorithms	51
Figure 21.	The Forward Step of the PEFC3 Algorithm	56
Figure 22.	Comparison of the Number of Consistency Checks for SMV, MMV, and PEFC3 Algorithms	59
Figure 23.	Percentage Difference from Optimal Costs for SMV and MMV Algorithms	60
Figure 24.	Schematic Diagram of The Activity Representation	69
Figure 25.	A Simple Precedence Constraint Between 2 Activities	70
Figure 26.	A Representation of Two Allen Relations with TemporalConstraints	70
Figure 27.	Schematic Diagram of The Activity Representation	71
Figure 28.	Constraint Representation of a Simple Job Shop Scheduling Problem	72
Figure 29.	Optimization Constraints on the Example Scheduling Problem	73
Figure 30.	Flow Chart of the General Estimation Algorithm	75
Figure 31.	The Average Cost Found by the Depth Algorithm with $L = 1$ for the Problem Sets with 1 Bottleneck	87

Figure 32. The Average Cost Found by the Depth Algorithm with $L = 1$ for the Problem Sets with 2 Bottlenecks 88
Figure 33. Raw Data for Problem 3, Problem Set 2 91
Figure 34. Raw Data for Problem 1, Problem Set 5 92
Figure 35. Raw Data for Problem 9 in Problem Set 8 92
Figure 36. The *EstGap* Statistic for Estimations using Different Texture Measurement 93
Figure 37. The *ChangeActivity* Count Estimations using Different Texture Measurement 94
Figure 38. The Effect of Subgraph Size on the *EstGap* Statistic 96
Figure 39. The Effect of Subgraph Size on the *ChangeActivity* Count 96
Figure 40. The Effect of Parameter Settings on the *EstGap* Statistic 97
Figure 41. The Effect of Parameter Settings on the *ChangeActivity* Count 98
Figure 42. The Best *EstGap* Statistic for Each Texture Measurement and Subgraph Size 99
Figure 43. The Lowest *ChangeActivity* Counts for Each Texture Measurement and Subgraph Size 99
Figure 44. Logistics Level Constraint Graph of the Example 108
Figure 45. Logistics Level Constraint Graph After Relaxation 110

List of Tables

TABLE 1. Variations of the SMV algorithm 53
TABLE 2. Variations of the MMV algorithm 53
TABLE 3. Variations of the MC algorithm 54
TABLE 4. Parameters of the Experimental Problems 57
TABLE 5. Average Number of Constraints and Average Optimal Cost for each Problem Set 63
TABLE 6. Variations of the Cost Estimation Algorithms 90
TABLE 7. Current Schedule in the Supply Chain Simulation 108
TABLE 8. Schedule After Relaxation 109

Chapter 1 Introduction

Constraint relaxation is the modification of the valid relationships among a set of variables in a constraint-based representation of a problem. The modification changes the problem definition, allowing a superset of the original solutions. Its chief application is where finding a solution to the original problem is prohibitively expensive or infeasible. Relaxation has the potential for a much larger role in constraint-based search and reasoning than it has played to this point. We are motivated by applications in guiding general constraint satisfaction, scheduling, and coordinating autonomous agents. In this dissertation, we propose a schema for constraint relaxation based on the propagation of information through a constraint graph.

This chapter presents an informal description of constraint relaxation and examines our motivation. We then present the necessary background on the constraint satisfaction techniques from which some of the constraint relaxation techniques are evolved. Finally, we present a definition of constraint relaxation and contrast it to similar notions appearing in the literature.

1.1 Constraint Relaxation Overview

Constraint-based techniques represent problems as a set of variables and a set of constraints. Each variable has a *domain* of possible values and each constraint limits the values that a subset of the variables can take on. A constraint is the embodiment of a particular relationship among a set of variables. A simple example of a constraint-based representation is used in the graph coloring problem [Garey 79] . Each variable can be assigned a color from a limited spectrum and each constraint defines that a particular pair of variables can not have the same color. The variables in a pair that is so limited are directly *connected* by a constraint or, equivalently, are *adjacent* variables. Adjacency is specified in the problem definition. The goal in constraint satisfaction problems (and graph coloring, in particular) is to assign values (specific colors) to variables such that all constraints are *satisfied*; that is, so that the relationship defined by each the constraint holds. As reviewed below, the typical solution techniques search via the systematic assignment of values to variables.

Constraint relaxation modifies the relationship defined by a constraint. As the term "relaxation" implies, the modification allows a wider range of relationships. In graph coloring, for example, we might decide that a particular adjacent variable pair can be different colors or can both be red. This highlights the difference between relaxation of a constraint and non-satisfaction. Relaxation makes a particular change in the definition of the constraint (e.g. allowing both variables to be red), whereas non-satisfaction removes the constraint completely: there is

no limit on the values to which the variables can be assigned.[1] Once a constraint is relaxed, the problem is altered because a relationship that was not allowed in the original problem is now acceptable.[2]

There has been some work on the selective non-satisfaction of constraints [Freuder 92] , and other work that can be interpreted as making contributions toward relaxation [Rosenfeld 76] [Dechter 90] . A well-grounded theory is lacking. Such a theory has wide applications to any problem expressible in the constraint satisfaction paradigm.

1.2 Motivation

Constraint relaxation is applicable in two general areas. The first is in guiding search for a solution to an original problem. If we relax the problem, we may be able to use the easily found solutions to focus the search in the original problem. Secondly, relaxation is necessary in *overconstrained* situations, where it is impossible to assign values in such a way that all constraints are satisfied.

Our chief motivation for this work is the use of constraint relaxation as a coordination mechanism in multiagent domains. We plan to investigate coordination in the domain of the supply chain for a manufacturing enterprise. We have a number of additional motivations in the areas of guidance of general search, scheduling, and a theory of constraint-directed reasoning.

1.2.1 Coordination of Multiple Agents

Our foremost interest in constraint relaxation is in its use as a mechanism for efficient cooperation among a group of autonomous, resource-sharing, problem solving agents as they attempt to meet their own and group ends. This area of *multiagent coordination* has received a great deal of interest in the field of Distributed Artificial Intelligence [Distributed 87] [Distributed 89] . We have adopted *supply chain management* as the focus of our work in coordination because of the close mapping between the departments and actors in the supply chain and agents in a shared software environment. This mapping is seen in the fact that departments within an enterprise work towards both global and local goals with shared, finite resources. Furthermore, often departments must work together to achieve these goals. We use the supply chain, here to highlight many of the challenges surrounding the coordination of agents.

In a manufacturing enterprise, the supply chain consists of all activities leading to the delivery of a product to a customer. These include research and development, marketing, accounting, material planning, production planning, production control, transportation, and customer service. Figure 1 represents the flow of material in the supply chain of a manufacturing enter-

---

1. We view non-satisfaction of a constraint as *complete* relaxation because it changes the relationship enforced by a constraint to the point where any value pair will satisfy the constraint.

2. By specifying a subset of constraints as nonrelaxable, we can model problem with both “hard” (nonrelaxable) constraints and “soft” (relaxable at some cost) constraints.

prise. The diagram shows one aspect of the supply chain, leaving out a number of functions as well as a representation of the flow of information and feedback. Figure 1 illustrates that the supply chain is a complex interaction of a number of functional entities in order to achieve some level of performance in the delivery of products to customers.

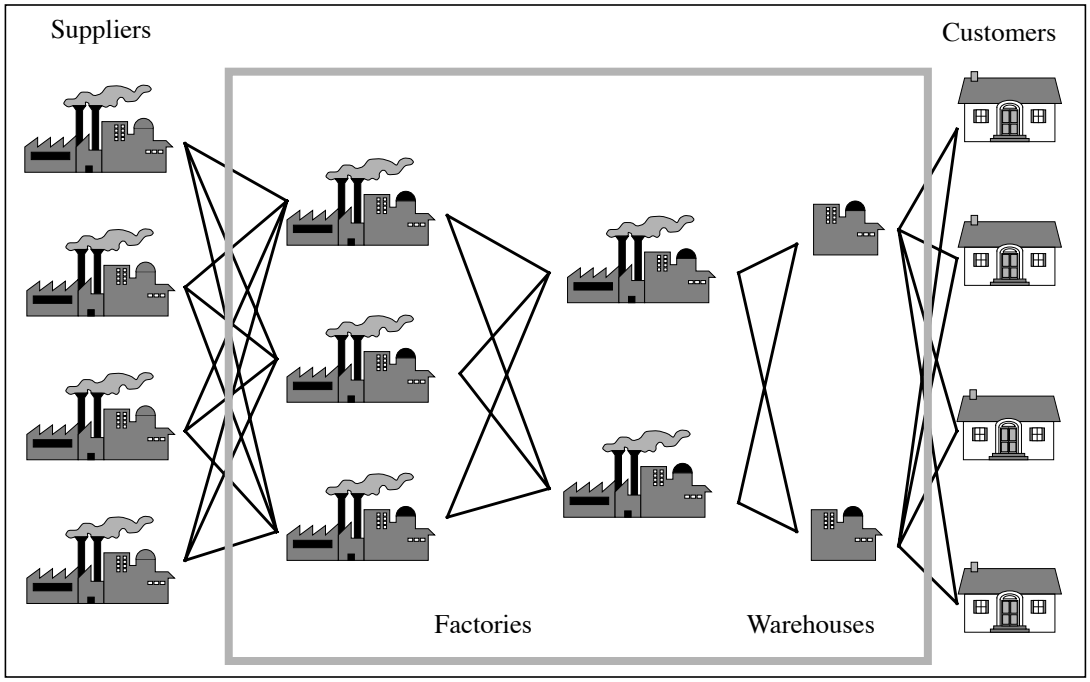

Figure 1. The Material Flow in the Supply Chain of a Manufacturing Enterprise

Each link in the chain is subject to unexpected events, and quick response to these events is a key element in the success and survival of a corporation [Nagel 91] . Exogenous events are many and varied: change in the customer order, late delivery or price change of a particular resource, machine breakdown, an urgent order from a good customer, and so on. Handling these events requires close coordination and cooperation among all departments in the enterprise. The following example illustrates the scope of the problem [Fox 92] .

The Canfurn, Inc. furniture company produces furniture with various options. Leo's, the largest and best-paying customer of Canfurn, places a large order for delivery in six months and Canfurn schedules delivery. Two months before the original delivery date, Leo's requests a significant change in the order but wants to maintain the same delivery date. Canfurn's sales department immediately contacts the manufacturing division. Manufacturing has a number of options and a number of questions that need to be answered:

- Manufacture the modified order. Can the new order be manufactured? Are extra shifts needed? What does personnel think of extra shifts? Are the raw materials in stock? If not, can a supplier be found?
- Delay another order. Can another order be delayed (and delivered late) in order to meet Leo's order? What does sales think?

- Subcontract the job to another manufacturer. Can the job be subcontracted? What does marketing and strategic planning think? What does accounting say about reducing the margins? Can we afford to take a loss on the order?

Clearly, the manufacturing division cannot make the decision independently. It must canvas a number of other divisions within the company and some external bodies (suppliers and subcontractors) in order to choose an alternative that is as inexpensive as possible.

The Enterprise Integration Laboratory at the University of Toronto is developing an Integrated Supply Chain Management System (ISCM) addressing these and other problems. The project is based on a distributed simulation of an enterprise, where departments are encapsulated as software agents. Given this distribution, the inter-departmental coordination in real corporations is manifest in the inter-agent coordination in the simulation.

The inter-agent coordination is hierarchical. With multiple production centers (e.g. factories), coordination among agents within one center is a level of abstraction below the coordination among the centers. The latter abstraction represents enterprise-wide logistics. It has a global view of the enterprise and is concerned with sales, delivery to the customer, and all aspects of inter-production-center coordination.

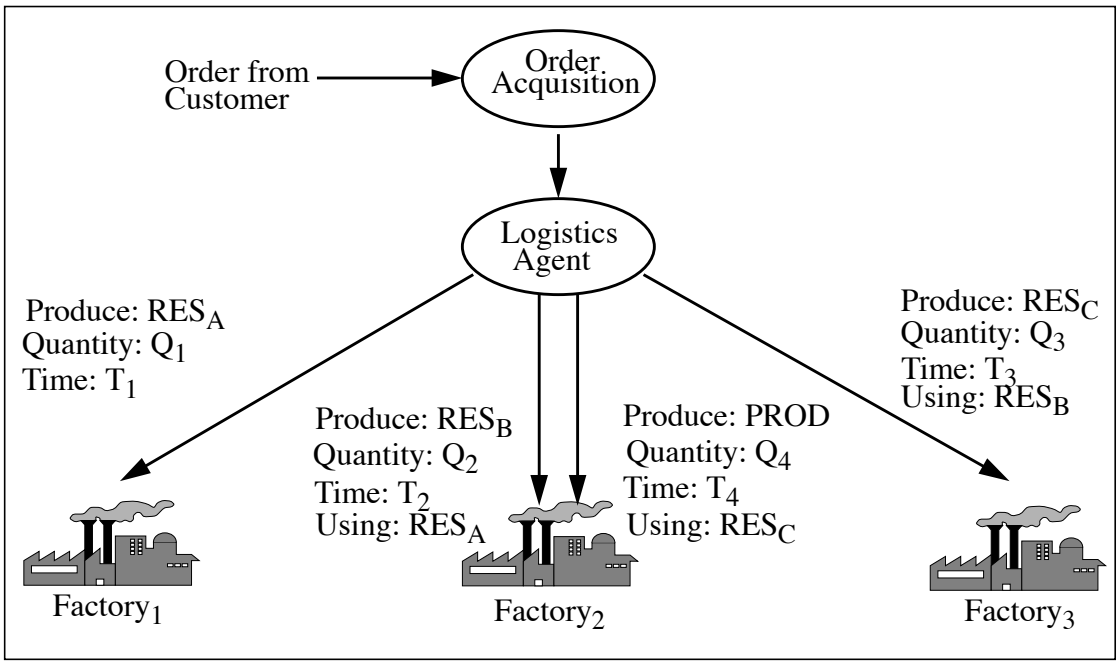

Figure 2. Logistics Level Scheduling of a Customer Order

Each production center is viewed as a single resource with the ability to perform multiple activities resulting in the production of a quantity of a resource. The activities each factory can perform and the capacity of each factory is known.[3] Logistics-level scheduling assigns factories to supply specific quantities of resources at particular times.[4] The factories commit to these assignments. Figure 2 shows a schematic of the logistics-level assignments when an

order is received via the Order Acquisition agent. On the basis of these commitments, activities at other production centers are scheduled. This results in a *commitment graph* of interdependent activities such as shown in Figure 3.

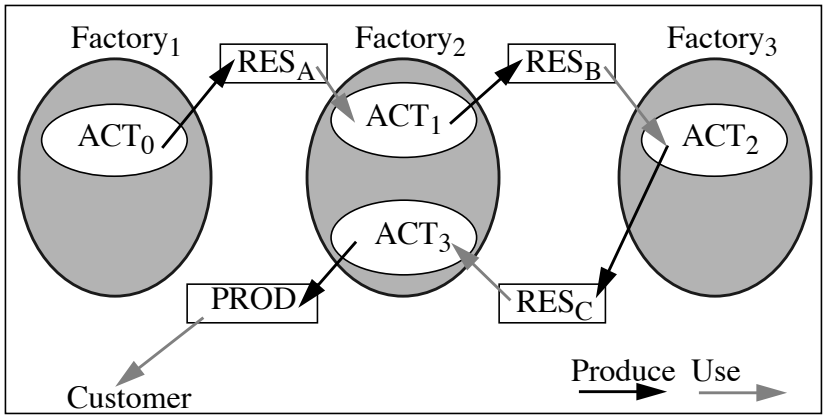

Figure 3. A Simple Activity/Resource Constraint Graph

With the occurrence of an unpredicted event, it is likely that factories will not be able to meet all their commitments. In these infeasible situations, it is necessary to assess the alternatives and adopt the one with a small negative global impact. The commitment graph must be reconfigured to escape an overconstrained situation: this is constraint relaxation. Given multiple orders and activities at each factory, a full constraint graph will certainly grow to a non-trivial size. A search for a near-optimal reconfiguration has to handle the combinatorial explosion of interdependent alternatives in resource choice, transportation method, and execution times, to name just a few. We believe that constraint relaxation guided by structural properties of the commitment graph can be applied to this problem.

1.2.2 Guiding Search with Relaxation

Problem solving can be guided by solutions to related, simpler problems. This technique has been used in the Operations Research community in the branch-and-bound technique [Hillier 80] [Cplex 92] . The solution space is partitioned based on a bound on the optimization function. The bound is found by solving a problem that ignores several of the original constraints. Portions of the solution space are selected and further partitioned based on assertion of one of the original constraints. Iteration continues until all constraints have been asserted and a solution is found or it is found that no feasible solutions exists.

---

3. These capacities represent aggregate information based on previous performance. Environmental events (e.g. machine breakdown) can dynamically impact these capacities.

4. For now, we ignore the scheduling of the transportation between factories and the delivery to the customer, though this is included in the scope of the ISCM project.

The ISIS scheduling system uses constraint relaxation to rate a number of competing partial schedules [Fox 87] . Based on the rating that reflects the relaxation necessitated by the partial solution, a subset of the solutions are selected for extension.

It has been shown that the shape or *topology* of the constraint graph has significant effect on the efficiency of the search for a solution. In particular to ensure a linear time search, the graph must be tree-shaped [Freuder 85] . The ABT algorithm is created to exploit this condition [Dechter 88] . ABT finds the minimum spanning tree of the full constraint graph and exhaustively generates solutions to the spanning tree graph. The number of solutions that each value in the domain of a variable takes part in is used to guide the value ordering for subsequent search through the full constraint graph.

Using solutions to a relaxed problem to guide search of the full graph is intriguing. Each of the above examples uses solutions to relaxed problems to expedite search in the full problem. A firm theory of constraint relaxation may point toward relaxation schemes that produce easily solved problems in many different problem solving contexts.

1.2.3 Scheduling

Scheduling is a difficult task that can be formulated using a constraint representation. Reasons for scheduling difficulty include [Fox 90a]:

- Scheduling is a constrained optimization problem over such criteria as tardiness, process time, inventory time, and cost. The problem space can be very large.[5]
- Many scheduling problems are overconstrained due to the unavailability of resources given the temporal constraints (e.g. due dates, release dates, and precedence constraints).
- Constraint representations have failed to stress the importance of domain values. The number and identity of tasks that require a resource over a particular time interval is a key piece of information that can form the basis for heuristic variable and value orderings.

Underlying these difficulties is the fact that many of the constraints in scheduling are disequality constraints (e.g. two tasks cannot use the same resource at the same time). Disequality constraints create the large search space that may have few (or no) satisfying solutions and make necessary the close attention to domain values.

That scheduling is actually a constrained optimization problem makes it a candidate for relaxation. If the optimization criteria are explicitly represented by relaxable constraints, schedule optimization is equivalent to finding a low cost relaxation of the optimization constraints. Furthermore, since scheduling problems are often overconstrained, relaxation is sometimes needed to find any solution, not just optimized solutions.

---

5. For example, a 100-activity problem, with 100 time slots and 100 resources has a search space of 10^{400} [Fox 90a].

1.2.4 Constraint-Directed Reasoning

A further motivation for addressing constraint relaxation is the belief that constraints have a large role in heuristic problem solving. A theory of constraint-directed reasoning points toward constraints and the constraint graph structure as the main features in search. The ISIS system [Fox 86] [Fox 87] uses a rich constraint representation in extending the heuristic search paradigm to include constraint-directed reasoning. Not only do constraints define admissible solutions, but also they establish a partial ordering among those solutions, generate alternate search states, and provide a structure for the problem solving knowledge. [Fox 86] presents eleven observations of the role of constraints in problem solving. These roles include state-generation, parameterization of the search space, providing the contextual relevance of an operator, and stratification of both the representation and the search.

Relaxation is an important component of any theory of constraint-directed reasoning. In ISIS two modes of relaxation are used: analytic and generative. Analytic relaxation chooses constraint alternatives on the basis of an examination of the current state of the search. Generative relaxation uses constraint alternative choices as an operator to produce a new search state. For relaxation to play such roles, a firm semantics is needed to assess why and how a relaxation should be done. A range of tools that can be applied in both analytic and generative relaxation to suggest appropriate relaxations given the problem solving context is also required.

1.3 Constraint Satisfaction Problems (CSPs)

Before presenting a definition of constraint relaxation, we review the constraint satisfaction literature upon which constraint relaxation is based. In this section, we will present the typical constraint satisfaction model and discuss solution techniques.

1.3.1 The CSP Model

The general CSP consists of the following:

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

A solution to a CSP is an assignment of a value to each variable from its respective domain that satisfies all constraints. In the basic CSP, all solutions are equally rated.

The most common conceptualization of a CSP is as a graph with variables represented by nodes and constraints by arcs.[6] The graph structure leads to the notion of traversal of constraints and of "transmitting" information between variables over the constraint links. This transmission is called *propagation*. Propagation is an important mechanism in the relaxation algorithms presented in this dissertation.

The constraint model has proven to be a general problem solving structure employed in many areas. The following is a partial list of areas where the constraint formalism has been applied (adapted from [Kumar 92]). It reflects the broad applicability of the constraint-based approach to problem solving:

- Planning [Descotte 85] [Conry 86] [Lansky 88] [Conry 91] [Pope 92] .
- Scheduling [Fox 87] [Fox 89] [Sadeh 91] [Ijcai 93] [Davis 94] .
- Truth-maintenance [Huhns 91] .
- Machine vision [Waltz 75] [Rosenfeld 76] [Shapiro 81] .
- Temporal reasoning [Allen 83] .
- Graphical user interfaces [Borning 87] .
- Logic programming [Borning 88] [Van 89] .
- Configuration [Mittal 90] .
- Abduction and induction [Page 91] .
- Design [Navinchandra 87] [Navinchandra 91] .
- Enterprise modeling and design [Gruninger 94] .

1.3.2 CSP Algorithms

CSPs are amenable to generate-and-test algorithms where all possible value assignments are systematically evaluated. On average, however, more efficient search can be achieved when the variables are instantiated sequentially in some order. We follow [Freuder 92] in dividing the sequential solution techniques into retrospective and prospective flavours. An excellent survey of CSP solving is presented in [Kumar 92] .

1.3.2.1 Retrospective Techniques

Retrospective techniques are characterized by the assignment of a value to a variable followed by testing against variables that are already assigned. If all constraints are satisfied, another variable and value are selected. If some constraint is not satisfied, *backtracking* occurs. Backtracking reassigns one of the variables to another value in its domain. If all values in the domain of a variable have been unsuccessfully tried, the variable is left unassigned and backtracking moves to another variable.

---

6. Other conceptualizations exist, including the dual constraint graph and join graph [Dechter 90] . In general, a constraint graph is a hypergraph with arcs connecting all relevant nodes.

The simplest retrospective technique is *chronological backtracking*. If the compatibility check fails, another value is tried for the most recently assigned variable. If all values in the domain of the variable are unsuccessfully tried, the variable is left unassigned and a new value is assigned to the next-most recently assigned variable. The gain over generate-and-test comes when an incompatibility is found amongst a subset of variables before all variables are instantiated. That part of the search space is pruned because there is no possibility of a solution.

Chronological backtracking suffers from a number of problems, in particular inefficient behaviour known as *thrashing* [Mackworth 77] [Kumar 92] . Thrashing occurs when the search fails in multiple places for the same reason. If all the values for a variable fail, chronological backtracking returns to the most recently instantiated variable. It is possible, however, that the reason for failure is an inconsistency with a variable much further back in the search tree. A great deal of instantiation and backtracking will be done on the intervening variables before the value of the offending variable is changed. If more effort is expended on finding the cause of the failure, some of this wasted work may be avoided.

In both *backjumping* [Gaschnig 77] and *backmarking* [Gaschnig 78] information is cached at each variable to aid in the more precise identification of the offending variables. Instead of returning to the next-most recently assigned variable upon a failure, a number of variables are backtracked through at one time to find the offending variable. Search is significantly reduced by avoiding instantiating the value combinations of the intervening variables. The difference in the two methods is the amount of information kept at each variable. Backmarking stores more information and so is more efficient in terms of the number of compatibility checks it needs to do. Because backmarking requires storage of a larger amount of information at each variable, the space complexity is better for backjumping.

An additional inefficiency in backtracking is the performance of redundant work. For example, consider two variables, x_a and x_b, that are widely separated in the variable ordering yet share a tight constraint. If there is a complex relationship amongst the intervening variables, a non-trivial amount of work is done in discovering this relationship. If the tight constraint causes a failure at x_b, backtracking will return to x_a. The relationship amongst the intervening variables must be re-discovered when search proceeds forward with a new value for x_a. Chronological backtracking, backmarking, and backjumping suffer from this fault because they uninstantiate intervening variables when returning to the failure-causing variable. *Dependency-directed backtracking* addresses this problem [Stallman 77] [Kumar 92] . In general, when an inconsistency is found, it is recorded and used as a justification for a new assignment. If backtracking occurs past these variables, the justifications still exist and an assignment can be made without recapitulating the previous search. If dependency-directed backtracking is fully implemented all redundant work is avoided. The complexity of finding and maintaining the justifications is so high, however, that often dependency-directed backtracking performs worse than chronological backtracking [Kumar 92] .

1.3.2.2 Prospective Techniques

In contrast to retrospective techniques, prospective CSP solvers propagate the effects of a variable instantiation to *unassigned* variables. This propagation is based on three *consistency algorithms* that grew from a recognition of three reasons for thrashing [Mackworth 77] . The first reason is lack of *node consistency*: when elements in a variable's domain do not satisfy unary constraints. Assigning these values causes immediate failure. However when a non-node consistent variable is backtracked over and re-tried, the same failures will occur again. The second reason, lack of *arc consistency*, applies the same concept to binary constraints. If the value of one variable has no consistent counterpart in the domain of a directly connected variable, failure will eventually occur. When the two variables are far apart in the order of instantiation a great deal of work is done and no failure occurs until the latter variable is reached. This work is repeated multiple times until the backtracking finally returns to the first variable. Finally, a lack of *path consistency* is also a source of thrashing. With three variables, x_i, x_j, and x_k, where each shares a constraint with the other two, it is possible that for some pair of values for (x_i, x_j) there is no consistent value for x_k. This creates behaviour similar to that seen with lack of arc consistency. Consistency algorithms [Mackworth 77] have been defined that establish each of these conditions by the propagation of domain values across constraints. Unfortunately, except in special cases [Freuder 85] , establishing all or any of these forms of consistency does not guarantee a backtrack-free search. Prospective algorithms work by establishing some form of consistency after every variable instantiation.

The most common form of prospective algorithm is *forward checking* [Haralick 80] [Shapiro 81] . The algorithm instantiates a variable, x, and prunes inconsistent values from the domains of adjacent variables. If any of the domains become empty, failure occurs, and x must be re-assigned. When backtracking, care must be taken to return the domains to the state prior to the pruning. Early in the search, forward checking does more consistency checks than a retrospective algorithm, however, when domains have been pruned, many fewer consistency checks are performed.

If establishing some level of consistency is useful, it may be worthwhile to create a more fully consistent graph. Experiments comparing backtracking searches with different consistency algorithms between variable instantiation steps have been run [Nadel 88] [Kumar 92] . Results indicate that it is often better not to establish full consistency since the effort invested in achieving consistency can be greater than that necessitated by the additional backtracking.[7]

1.3.2.3 Variable and Value Ordering

In a backtracking search, significant performance changes can result from a modification to the order in which variables are instantiated and/or values are tried at each variable [Kumar 92] . In a CSP that has a satisfying solution, a perfect variable/value order would produce a linear time solution because no backtracking would be necessary. The most common heuristic

7. But see [Nuitjen 93] for some interesting evidence to the contrary in the scheduling domain.

is to instantiate the *most constrained* variable to its *least constraining* value. Intuitively, by instantiating the most constrained variable earlier, backtracking will take place earlier and thrashing can be minimized by pruning the search space. By assigning the least constraining value, the likelihood of finding a solution without having to backtrack to this variable is increased. Reliable identification of critical variables and values is an open research question.

1.3.2.4 Texture Measurements

None of the presented problem solving techniques escape a worst-case exponential search. As a consequence, it is important to make good heuristic search decisions in solving practical problems. In particular, *texture measurements* [Fox 89] can be used as a basis for heuristic decisions. A texture measure is an assessment of properties of a constraint graph reflecting the intrinsic structure of a particular problem. Examples include variable and constraint tightness and constraint reliance. Texture measures have been used in the job shop scheduling domain [Sadeh 91] as a basis for opportunistic variable and value ordering decisions.

A goal of research on texture measures is to find correlations between particular graph structures and performance of an algorithm [Fox 89] [Davis 94] . If such correlations can be found, the search algorithm could be automatically configured based on texture measurements. Such general results have not yet been shown.

1.4 Constraint Relaxation

Constraint relaxation is the modification of the meaning of a constraint that changes the relationship that it enforces among a subset of problem variables. Given the usual CSP model, relaxation of constraint, c_k, changes the predicate $c_k(x_i, \ldots, x_j)$, resulting in the predicate having a value of TRUE for larger subset of the Cartesian product of the relevant variables.

Because constraint relaxation changes the problem definition, a solution to a problem with some relaxed constraints is different from a solution to the original problem. We represent this difference by specifying that a cost is incurred when a constraint is relaxed. This cost depends on the specific modification that is performed because *a constraint can be relaxed in numerous ways*. The specific relaxation that is performed has significant impact on both the local cost of the relaxation and on its utility in later search.

A solution to a constraint relaxation problem is the assignment of values to variables and the relaxation of some constraints so that all constraints are satisfied and the cost is as small as possible.

1.4.1 Relaxation as Repair and Optimization

We have two views of constraint relaxation:

1. Relaxation is an operator in repair-based search. When a partially consistent solution exists for a CSP, we can easily find a solution to the relaxed problem by relaxing the constraints that are not satisfied. An attempt is made to minimize the cost by relaxing some constraints and changing some values. Relaxation can significantly expand the solution space of a problem and lead to subsequent search based on CSP techniques.
2. Relaxation is a form of optimization. We can model the optimization criteria as a set of relaxable constraints representing the sole source of cost in the problem.

These two views of relaxation are not incompatible. Especially in the scheduling domain, relaxation allows the modeling of problems where not only are there optimization criteria such as tardiness, but there are also options and trade-offs involved with problem constraints.

1.4.2 Relaxation and Selective Non-Satisfaction of Constraints

Selective non-satisfaction of constraints ignores some of the problem constraints in order to make the problem easier to solve. Ignoring a constraint means treating it as if it does not exist: it no longer places any limit on the relationships among the variables. We saw this in the ABT algorithm where all the constraints that were not in the minimum spanning tree of the graph are ignored. Non-satisfaction of constraints is also the main mechanism behind Partial Constraint Satisfaction Problems (PCSPs) [Freuder 92] . The goal in PCSPs is to find a valuation for the variables that satisfies as many constraints as possible.

Algorithms based on selective non-satisfaction of constraints fail to represent the semantic content of constraint relaxation. There are many ways to relax a constraint and each relaxation can have different effects on the relationship of the relaxed problem to the original. For example, in a scheduling problem, choosing to ignore a due date constraint may lead to a simple solution to the rest of the problem. However, relaxation of a due date constraint by 3 hours has significantly less impact than relaxation of the constraint by 3 days. Non-satisfaction of a constraint misses this point because it does not represent the difference between a relaxation of 3 hours and a relaxation of 3 days.

Despite the fact that non-satisfaction of constraints does not represent the key semantic issues, it is a special case of constraint relaxation. In our model of constraint relaxation, the function that finds the cost of a relaxation of a constraint can be specified. We can define that the cost of all relaxation of all constraints is equal and therefore model non-satisfaction of constraints.

1.5 Contributions of this Work

The contributions of this dissertation are fivefold:

1. A generalization of the constraint model that allows generation of alternative constraints and an assessment of their local impact. This generalization represents the semantic meaning of relaxation by defining the cost of relaxation to be based on the constraint itself and on how relaxation is done. The ability to generate and assess alternative constraints is a significant addition to general constraint techniques.
2. A flexible relaxation schema within which relaxation techniques can be specified and investigated. The schema is applicable to all types of constraints and is modulated by heuristic decisions specified at the time of instantiation.
3. A specification of three relaxation algorithms shown to perform well on Partial Constraint Satisfaction Problems (PCSPs). These algorithms demonstrate the applicability of algorithms defined within the schema to repair-based search on general constraint graphs.
4. The application of propagation techniques and texture-based heuristics within the relaxation schema to schedule optimization. We model the sources of cost in schedule optimization with relaxable constraints. Three schema-based algorithms are defined which assess the impact of scheduling decisions by estimating the relaxation necessitated by the decision.
5. The modeling of multiagent coordination as a problem requiring constraint relaxation. We represent the interactions among agents as a shared constraint graph, and therefore, a situation requiring coordination is manifest by the need to re-establish feasibility in this constraint graph. Relaxation is a technique to re-establish graph feasibility.

1.6 Plan of Dissertation

In Chapter 2, we review the literature relevant to constraint relaxation. We present our propagation-based relaxation schema in Chapter 3. Three instantiations of the relaxation schema for Partial Constraint Satisfaction Problems (PCSPs) are specified and empirically evaluated in Chapter 4. In Chapter 5, we turn to relaxation for schedule optimization. We describe the job shop schedule optimization problem and present three algorithms for the estimation of the impact of scheduling decisions. Chapter 5 also presents empirical results comparing the estimations made by our algorithm to baseline estimates. In Chapter 6, we return to the domain of multiagent coordination and discuss the relevance of the relaxation techniques developed in this dissertation. Finally, in Chapter 7, we conclude and a look at prospects for future work.

Chapter 2 Related Work

In this chapter we present background for the relaxation problem. We examine probabilistic labeling, an early form of constraint relaxation used in machine vision. This is followed by a discussion of placing weights on constraints to indicate relative importance and some recent work on partial constraint satisfaction. Since the relaxation can be used for constraint optimization, we briefly review work in this area. We conclude with a review of constraint-based scheduling work.

2.1 Probabilistic Labeling in Machine Vision[1]

A common problem in machine vision is to label objects appearing in an image. Given an *a priori* set of objects and possible spatial relations among them, the problem can be represented as a CSP where the goal is to label the objects such that the relations are satisfied. Most of the approaches to the labeling problem use a form of arc consistency algorithm first suggested by [Waltz 75] . When no wholly consistent labeling can be found (e.g. due to uncertain data), it is necessary to find a probabilistic labeling by allowing some spatial relationships to be violated.

[Rosenfeld 76] presents a probabilistic labeling algorithm that uses a weight vector at each object. Each weight corresponds to a label and is interpreted as the probability that the label is correct for that object. A function is defined that updates the weight of a label based on its likelihood of co-occurrence with other labels. A linear update function can be used with a vector of compatibility factors that defines the probability that a tuple is a correct example of each of the spatial relationships. The compatibility factors are interpreted as conditional probabilities. The algorithm updates the probability of each label at each object based on the compatibility of that label with more or less probable labels of other objects. The desired behaviour is to decrease the entropy of the labeling (i.e. decrease the uncertainty as to the label of each object). Ideally, the probability of a label is increased if it has a highly probable compatibility with highly probable labels and decreased if it has a low probability of compatibility with highly probable labels. It is shown that, with a linear update function, the label probabilities will converge on a fixed-point dependent only on the compatibility factors.

1. In the computer vision literature, "relaxation" refers to the convergence of a network to an arc consistent set of labels. This is different from our notion of constraint relaxation so we do not use "relaxation" in the context of the vision algorithms.

If the compatibility factors are defined on the interval [-1, 1] and interpreted as correlations (or covariances), it is necessary to use a nonlinear updating function to ensure that label probabilities are mapped to probabilities. Experimental evidence suggests that a solution can be found that takes into account the original label probabilities rather than converging on a fixed point. Later work refined this analysis by identifying an optimal updating function [Cooper 92] .

There are three undesirable attributes of these algorithms that discourage application to constraint relaxation: the process is subject to a bias, the calculated probabilities are hard to interpret, and the algorithms generally converge on a local extremum of the updating function [Cooper 92] . If all labels initially have the same probability, an unbiased algorithm will not change these probabilities. A constraint relaxation application of the algorithm would have uniform initial probabilities and so, by using the algorithm, we would either get a biased result (if the function is biased) or no change (if the function is unbiased). The convergence to a local extremum is troublesome because it is the global optimum that is desired.

2.2 Non-Satisfaction of Constraints By Weight

A simple form of non-satisfaction of constraints uses a weight at each constraint reflecting its importance or the preference for its satisfaction. The higher the weight, the more important it is to satisfy the constraint. In attempting to find a solution where some constraints do not have to be satisfied, the algorithm resolves conflicts by ignoring the constraint with lower weight. Weights can also be interpreted as a penalty or cost incurred if the constraint is not satisfied [Zweben 94] .

Naïvely, weights can be combined with a complete CSP solver that ignores constraints with weight below a threshold. The threshold is set to 0 and incremented each time the CSP solver fails. By storing information from previous iterations, significant performance gains can be made over attempting to solve the CSP many times. This has been demonstrated in a distributed application [Yokoo 93] and in the scheduling domain [Bakker 93] .

In contrast, [Descotte 85] integrates weights into the search in a process planning application where a plan for the machining of parts is to be produced from drawings. The quality of the machining, positioning of raw materials, and the actual machines for each action are subject to constraints and, in turn, produce constraints on subsequent steps. The main algorithm defines a number of constraints with preconditions and attempts to match the current partial plans to the preconditions. If a match is found, all corresponding constraints are applied to the plan in decreasing order of weight. Conflicts are resolved by selective backtracking: the set of active constraints with lowest weight is found and the most recently applied constraint is rejected. Iterative application of constraints and matching with preconditions then continues. The output is a set of plans that is guaranteed to be either empty (if no admissible plans exist[2]) or optimal. An optimal plan is an admissible plan in which the largest weight of an unsatisfied constraint is W, and there exist no plans such that the largest weight of a unsatisfied constraint is less than W.

The use of weights to indicate constraint importance can be generalized to a hierarchy of constraints [Borning 87] [Borning 88] . The top level of the hierarchy contains required constraints while lower levels contain constraints of decreasing priority. In addition to a weight, each constraint has an error function that provides a measure of the magnitude of dis-satisfaction. In a graphical user interface application, the algorithm takes the constraint hierarchy and attempts to assign values to parts of graphical objects so that all constraints are satisfied. If unsuccessful, the algorithm iteratively removes the lowest level of the hierarchy until the remaining constraints are satisfied or the hierarchy is empty. If the loop has removed some levels from the hierarchy, the most recently removed level is added to the hierarchy again. Objects affected only by constraints at this level are then modified to conform to the constraints at this level.

In the ISIS scheduling system [Fox 87] , it is recognized that the relative importance of constraints can vary widely with differing problem solving contexts. It is necessary to be able to establish and compare this relative importance. Three methods are discussed:

- Manual - the user can vary the importance measures from order to order.
- Partitioning - a certain class of orders may have a characteristic constraint importance pattern. This pattern can be automatically applied when the order class is known.
- Relational - relations are used to establish an importance ordering among the constraints.

ISIS needs to know not only the order of importance among constraints but also the magnitude in the difference in importance. Relational importance ordering does not address the issue of magnitude and, therefore, only manual and partitioning methods are used.

Because ISIS attempts to satisfy many constraints concurrently, it needs a basis of comparison of valuations. The importance measure is used as a weight and combined with the constraint utility to form this basis. The relative quality of valuations are assessed by measuring the weighted sum of utilities.

2.3 Extending CSP Solving Methods

There has been a great deal of work on solving CSPs and a number of algorithms have been proposed. Since constraint relaxation is an extension of constraint satisfaction, the algorithms proposed for CSPs may be extendible to constraint relaxation. In the previous section, we reviewed work on a weak version of relaxation using CSP solvers that ignore all constraints below a threshold. Here we focus on work that more deeply integrates selective non-satisfaction of constraints into the search.

---

2. [Descotte 85] defines a constraint with weight 10 as required: no admissible plan can contradict it. Declaring a subset of the constraints as required (or “hard”) has been used in other work [Fox 87] [Borning 87] [Borning 88] .

The key modification to the usual CSP algorithms is a redefinition of failure. In forward checking, a value is pruned from the domain of an uninstantiated variable if it is inconsistent with a current assignment. Work in extending CSP methods to include non-satisfaction [Shapiro 81] [Freuder 92] employs a threshold such that no solution can have a cost greater than the threshold. The *inconsistency count* is the local cost that would be incurred if a specific value of a variable is assigned. A cache is kept for each value of each variable and originally set to 0. When a value is assigned, a modified consistency algorithm propagates the value to adjacent variables. The cost of each value in the domain of an adjacent variable is incremented if it is inconsistent. The sum of current cost of all instantiated variables, the cost of the value, and perhaps other cost estimates (see below) is compared to the threshold. If it is greater than or equal to the threshold, the value is pruned from the domain of the variable. If the domain becomes empty, backtracking occurs.

```
assign a value to a variable and update CostSoFar
for all uninstantiated, connected variables{
   for all values in the domain{
      if CostSoFar + InconsistencyCount + Sum of minimum
         InconsistencyCounts from uninstantiated variables ≥
bound{
         prune
         if domain is empty backtrack
      }
   }
   for all remaining values in the domain{
      update InconsistencyCount
      if updated and CostSoFar + InconsistencyCount + Sum of min.
         InconsistencyCounts from uninstantiated variables ≥
bound{
         prune
         if domain is empty backtrack
      }
```

Figure 4. The Forward Step of the PEFC3 Algorithm [Freuder 92]

For example, the partial extended forward checking algorithm (PEFC3) [Freuder 92] [3] estimates a lower bound on the cost of assigning a value by summing the cost of the solution so far with the inconsistency count of the value and the sum of the minimum inconsistency counts for the uninstantiated variables. Intuitively, the last term is the minimum further increase in cost that choosing the value would incur. If this sum equals or exceeds the cost of the best complete solution found so far, the value is pruned from the domain of the variable. Backtracking occurs when a domain is empty. Pseudocode for the forward-step of the PEFC3 algorithm is shown in Figure 4.

3. PEFC3 is equivalent to the forward checking algorithm presented by [Shapiro 81] .

Four algorithms are proposed by [Shapiro 81] for use in machine vision object labeling: normal backtracking, forward checking, lookahead by one, and lookahead by two. The algorithms use increasingly sophisticated mechanisms to estimate a lower bound on the cost of a label. The general form is a tree-search bounded by the cost threshold as described above. User-defined compatibility costs are assigned to each possible label tuple. Experiments using randomly generated labeling problems and the first three algorithms show that the forward checking algorithm performed best.

A more systematic extension of CSP algorithms is due to [Freuder 89] [Freuder 92] . In this Partial Constraint Satisfaction Problem (PCSP) work, each constraint incurs a cost of 1 if it is left unsatisfied. The problems are defined so that a solution where all constraints are satisfied does not exist. The goal is to find the valuation that fails to satisfy as few constraints as possible. Adaptations of simple backtracking, backjumping, backmarking, as well as arc consistency and a variety of forward checking techniques are developed. A large set of experiments is performed comparing these algorithms on a variety of randomly generated PCSPs and on a set of difficult graph coloring problems. Results indicate that these algorithms perform well on the wide variety of problems. PEFC3 was generally superior across all the problem types. We present a comparison of our relaxation algorithms to the PEFC3 algorithm in Chapter 4.

PCSPs have been addressed with a distributed partial constraint satisfaction algorithm [Ghedira 94] . Each variable and each constraint is modeled as an agent with simple behaviour. A variable chooses a value for itself, polls its constraints for satisfaction, and accepts or rejects the new state based on a simulated annealing approach. The asymptotic convergence of the algorithm is shown; however it is recognized that this convergence is impossible to obtain in practice. No experimental results are presented.

The GSAT algorithm [Selman 92] operates on a problem that is, at least superficially, similar to PCSPs. Given a set of propositional clauses with a random assignment of truth values, the GSAT procedure changes the truth value of the literal that leads to the largest increase in the total number of satisfied clauses. The flipping of truth values continues until a satisfying solution is found or a bound on the number of flips is met. The GSAT method, which employs a full one-step lookahead, is surprisingly effective at finding solutions. The difficultly with extending GSAT to PCSPs (or general constraint problems) is the very small domain for each variable (i.e. {TRUE, FALSE}). It is not clear that the GSAT technique, which is expensive in terms of the size of the possible successor states it investigates, can be extended to a broader range of problems.

2.4 Constraint Optimization

Constraint relaxation problems are properly a subset of constraint optimization problems. Constraint optimization attempts to optimize the value of a general function over the variables in the problem while constraint relaxation is optimization the sum of the costs of the constraints.

2.4.1 Optimization in Operations Research

Optimization has traditionally been the purview of Operations Research (OR) techniques, such as linear and integer programming. The mathematical programming techniques center around a matrix representation of the problem variables and problem solving consists of operations on the matrix. There are some difficulties in the application of these methods to optimization problems investigated in AI research.[4] These problems include the following:

- Techniques such as integer programming show a tendency to be overwhelmed by the number of variables and values needed to represent non-trivial problems [Van 89] [Fox 90] [Sadeh 91] [Dorn 92] . Furthermore, these techniques often place restrictions on the constraints and variables that are not justified by the problem definition.
- Mapping a problem into the OR representation often necessitates a substantial increase in the number of variables and constraints used in other representations [Van 89] . This further aggravates the tendency for OR techniques to be overwhelmed by large problems.
- Mathematical programming approaches are awkward when a concrete measure of optimality can not be formulated [Dorn 92] . Such situations can arise from complex and conflicting problem objectives.
- There is difficulty in dealing with uncertainty [Dorn 92] .
- Specific problem features can not be exploited;[5] nor can particular heuristics be used [Van 89] .

Sensitivity analysis is the sub-field of Operations Research concerned with modifications to a problem definition after the problem has been solved. There are methods that, given an optimal solution, can assess the impact of changes to the problem constraints [Phillips 76] [Bazaraa 90] . These methods do not deal with overconstrained situations and, furthermore, do not provide any methods to identify constraints that are critical to the infeasibility of the problem.

Sensitivity analysis, using the theory of duality, allows for the simple derivation of *shadow prices* once the original problem is solved [Hillier 80] [Ravindran 87] . In a scheduling problem, the shadow price is the net impact on the total profit of additional unit of a particular resource. These increments are only valid within a range of resource quantities as the purchase of more will change the identity of the optimal solution (calculated by the original OR method). We will return to shadow prices later, in the context of predicting the impact of scheduling decisions. For now, note that while shadow prices are inexpensively derived from the calculation of an optimal solution with OR methods, the original calculation is subject to the drawbacks discussed here. Additionally, the fact that the shadow prices are valid only within a range limits the generality with which alternative resource purchases can be explored.

---

4. Despite these difficulties, there is a growing realization that benefits would accrue from the marriage of AI and OR techniques [Lowry 92] [Interrante 93] .

5. We assert that utilization of problem features identified with texture measures is a fundamental technique in heuristic problem solving.

The OR branch-and-bound methodology discussed in Chapter 1 (Section 1.2.2) uses solutions to relaxed versions of the problem to bound search in the original. This is of little use when the original problem is infeasible, since there is no easily available information about critical constraints. Furthermore, it is typical that only certain types of constraints can be relaxed to find a bound (e.g. relax the requirement that a subset of variables have integer values).

2.4.2 Optimization in Artificial Intelligence

[Dechter 90] presents an optimization algorithm based on transformations of the constraint graph. The optimization function is defined as the sum of a number of sub-functions where each sub-function operates on a subset of the problem variables. The *containment criteria* requires that the set of variables in each sub-function must be constrained by at least one problem constraint. For example, we have a problem with variables $x_1, \ldots, x_5$ and the objective functions $f(x_1, \ldots, x_5) = f_1(x_1, x_3) + f_2(x_2, x_3, x_4) + f_3(x_5)$. The containment criteria is met by the constraints: $c_1(x_1, x_2, x_3, x_4)$ and $c_2(x_2, x_5)$: f_1 and f_2 are contained by c_1 and f_3 is contained by c_2.

The constraint graph is transformed into a *dual constraint graph*, where each constraint is a node and arcs are labeled with the variables shared between the constraints. In special cases, a *join-tree* [Dechter 89] can be constructed where each node represents a cluster of related variables and the children of a node contain a subset of the variables in the parent. It is shown that, if the containment criteria is met and a join-tree can be created, the optimal tuple for a node can be calculated by choosing the maximally-rated tuples from the children. Therefore, the optimization is a recurrence algorithm beginning with the leaves and flowing toward the root with a complexity linear in the number of constraints and tuples.

When the objective function does not meet the containment criteria and the graph does not have a join-tree, the constraint graph is augmented following a tree-clustering algorithm [Dechter 89] . The tree clustering adds constraints to the graph in order to establish the both the containment condition and the necessary requirements for the dual graph to have a join-tree. The complexity of the overall algorithm is dominated by the need to solve a CSP at each node of the dual constraint graph.

[Yokoo 92a] address constraint optimization in a distributed environment where each agent has one variable and is aware of all associated constraints. Each agent also has a local objective function defined over all variables in the problem. A modified form of the distributed asynchronous backtracking algorithm [Yokoo 92b] is used where agents choose a local value and notify adjacent agents according to an ordering of the agents. The modified algorithm allows each agent to employ its own cost threshold and discover when no solution is possible at the current threshold. In that case, the agent increments its local threshold and continues the search. The discovery and incrementing is done independently by each agent.[6]

---

6. The optimization algorithm is similar to the weight-based algorithm noted in Section 2.2 [Yokoo 93] .

2.5 Scheduling

Scheduling problems are constrained optimization problems that may not have even a satisficing solution [Fox 90a] [Fox 90b]. In cases where the schedule is overconstrained, we need relaxation to address this problem. In this section we will review work done in optimized scheduling and in scheduling in overconstrained situations.

In constructive scheduling, a schedule is formed by the sequential assignment of start times and resources to activities. If a consistent assignment cannot be made for an activity, backtracking is done. Repair-based scheduling, in contrast, begins with a schedule that breaks one or more of the problem constraints. By reassigning activity start times and resources, the search tries to find a valid schedule. Fuzzy logic has been used as a method of dealing with uncertainty and infeasibility.

2.5.1 Constructive Scheduling

The MICRO-BOSS scheduler [Sadeh 91] [Sadeh 94] builds on previous work in ISIS [Fox 87] and OPIS [Smith 89] [Fox 90b]. A key difficulty in production scheduling is dealing with bottleneck resources. These are resources which are required by a number of activities over a particular time interval. Earlier work [Smith 89] [Fox 90b] observed that bottleneck resources appeared and disappeared during scheduling depending on the scheduling decisions and the time interval under consideration. Because the bottlenecks have significant impact on the quality of the schedule, it is necessary to be able to detect and react to the emergence of new bottlenecks during the scheduling process. MICRO-BOSS takes an activity-centered perspective, allowing the focus of attention to be quickly shifted as the importance of activities change. This “micro-opportunistic” approach allows the dynamic identification of bottleneck resources which constitute important trade-offs and critical activities on those resources. Once identified, the critical activities are the focus of attention and, when the trade-offs are resolved, the rest of the problem is easier to solve.

Micro-opportunistic scheduling is applied to both satisficing scheduling and optimized scheduling. The latter employs a cost model based on tardiness and inventory costs. Identification of critical activities is done by use of texture measurements on the constraint graph representation of the scheduling problem. By aggregation of information about contention for a resource over some time interval, the activity most dependent on the resource for which there is the most contention can be identified. Time and resource reservations are then made on the basis of a related texture measure that estimates the cost incurred by the reservation. Empirical evidence shows that MICRO-BOSS outperforms the best of 39 combinations of priority dispatch rules and release policies [Sadeh 94] .

2.5.2 Repair-based Scheduling

Recently, scheduling systems have been built in the repair-based paradigm, where local heuristics are applied to improve the original inconsistent solution [Johnston 92] [Mcmahon 92] [Zweben 94] . This is of interest from a relaxation perspective because relaxation can be viewed as a repair-based process. An inconsistent solution implicitly performs constraint relaxations: if we accept the solution, we must relax the inconsistent constraints. Normal repair-based scheduling attempts to minimize the number of unsatisfied constraints by modifying values of the variables. A relaxation algorithm is an effort to modify *both* the constraints and the value instantiation to find a lower cost solution.

The MinConflicts algorithm [Minton 92] is representative of the repair-based algorithms. MinConflicts quickly finds an inconsistent starting solution and then iteratively chooses a conflicting task and re-schedules it to a time that will minimize the number of conflicts it has with other tasks. The partial lookahead procedure simply tries to schedule the chosen task at each possible time and evaluates the solution by counting the number of conflicts. The algorithm ends when no conflicts are left or a bound on the number of iterations has been met.

Despite the incompleteness of the approach, very good results have been achieved with theoretical problems such as N-queens[7] [Minton 92] and real-world problems such as scheduling observations for the Hubble Space Telescope [Johnston 92] .

2.5.3 Fuzzy Scheduling

An approach to both the constrained optimization nature of scheduling and uncertainty in the schedule modeling is using fuzzy logic.

[Dorn 92] applies relaxation based on fuzzy sets to a production process scheduling domain where a job places significant constraints on jobs that may precede or follow it. Fuzzy linguistic values are used to rank the importance of individual jobs and compatibility of every pair of jobs. Scheduling focuses attention first on the more important jobs. Typically, some jobs can not be scheduled without violating constraints and some adjacent jobs will have a low compatibility. A constraint that should be improved is found and the associated jobs are removed or exchanged with other jobs. An evaluation function is defined based on the fuzzy values of constraints and the search is heuristically guided on the basis of the evaluation function.

[Dubois 93] uses the fuzzy constraint model for job shop scheduling. The interpretation of fuzzy constraints can be preferences surrounding hard constraints or uncertainty in the environment. The release date and due date of orders, and the duration of tasks are modeled as fuzzy values. The precedence constraints are also modeled as fuzzy sets: a constraint can be satisfied to different degrees. Given these fuzzy sets, the authors define the global satisfaction level to be the smallest extent to which a solution satisfies a constraint. This introduces a total

7. Place N queens on an N-by-N chess board such that no queen can attack another.

ordering over potential solutions which is exploited in the solving paradigm. The search is a classical branch-and-bound depth-first search, interwoven with a propagation mechanism to ensure the consistency of the fuzzy temporal windows and a lookahead analysis on which decisions are based. The lookahead procedure is based on an estimate of the decrease in the satisfaction if the best choice is not made. This measure (which is used as the value ordering heuristic) is used as the variable ordering heuristic because it also estimates the "criticity" of the conflicts.[8] Early empirical results indicate that fuzzy constraints are more productive than crisp analysis as the flexibility and/or uncertainty of the problem can be captured with little increase in complexity. Further, it is asserted that the framework handles partially inconsistent problems and can be easily extended to deal with constraint priorities.

It is interesting to contrast this approach with the constraint representation used in the ISIS scheduler [Fox 87] . ISIS represents preferences by having the constraints express their acceptance or rejection of a solution on a 3-point scale. The required constraints define the admissible solutions while the preference constraints form a total ordering over all admissible solutions. Just as in the work of [Dubois 93] , the ranking of solutions allows the search to preferentially extend some solutions over others based on the satisfaction of the constraints. The fuzzy logic based satisfaction calculation finds the smallest extent to which any constraint is satisfied. The authors note that this satisfaction measurement means that the high degree of satisfaction of one constraint cannot counterbalance the low degree of satisfaction of another as is seen in ISIS. In other words, the fuzzy scheduler would prefer a solution where all constraints are satisfied at a medium level over a solution where some constraints are highly satisfied and others only satisfied at a low level. ISIS, in contrast, prefers solutions where the "sum" of the satisfaction is higher. Unfortunately, the critical question of which of these preferences leads to better schedules remains open and is probably dependent upon problem specific and problem domain specific factors.

2.5.4 Estimating the Impact of a Scheduling Decision

Many scheduling techniques (including those reviewed above) attempt to estimate the impact of assigning a particular start time to an activity. Because of the number of constraints among activities, the impact of a particular assignment is not fully known until all other activities are assigned and the entire schedule can be evaluated. This is not helpful when the goal is to create the schedule in the first place. What is desirable, then, is an inexpensive ranking of possible start times for an activity by an estimation of the impact of each one on the overall cost (or feasibility) of the schedule. This estimation can form the basis for a value ordering heuristic at each activity. A variety of static value orderings are possible based on dispatch rules, such as always trying the values from earliest to latest start time. However, work on dynamic estimations [Sadeh 91] , based on information gathered from the current partially constructed schedule, shows that improvements can be made over the dispatch rules.

---

8. The notions of criticity and reduction in satisfaction are similar to criticality of activities and cost increases used in MICRO-BOSS [Sadeh 91] .

The MRP-STAR [Morton 86] planning module is a system designed for Manufacturing Resource Planning (MRP) and, as such, estimates the cost of sequencing jobs through a number of (aggregate) workcenters. The systems models costs with scheduling costs (costs of raw material and costs of using a workcenter, less immediate revenues received) and tardiness costs (one-time cost penalty, accruing cost penalty, lost revenue, and lost good-will). This total cost of formulated and, in the analysis, the derivative is taken. This derivative indicates the marginal cost of increasing lateness of a job. The authors referred to these costs as *shadow prices*, leveraging off the similarity to the OR concept of shadow prices (see Section 2.4.1).[9] Calculation of the dynamic, or "shadow" price of a set of jobs at any given time is done by an iterative calculation beginning with some reasonable sequencing rule and using the derivative of the total cost. On the basis of the calculation, the jobs are resequenced and the iterative procedure continues. The shadow prices take into account not only the costs within an order, but also intra-order costs in estimating the impact of scheduling decisions. The authors note, however, that this computation can be expensive, there is a need to recalculate periodically, and the prices are somewhat unstable.

The GERRY scheduler [Zweben 92] [Zweben 94] , a repair-based system much like the MinConflicts algorithm, uses the number of conflicts that an activity will have if it is assigned to a particular start time as an estimation of the global impact. The intuition is that the number of local conflicts is a good estimator for the number of global violations. The results of GERRY demonstrates that the intuition holds in a number of typical scheduling problems.

The MICRO-BOSS scheduling system uses the propagation of marginal costs within an order to estimate the cost incurred by an assignment. Efficient methods for identifying and updating both the best start times and the marginal cost factors for each activity are implemented. The calculated estimate is shown to be useful in scheduling, however two major trade-offs are made to achieve efficient computation. First, the methods are based on the particular cost model that is used in the experimental problems. A different cost model requires a new algorithm to identify the best possible reservations for an activity. Secondly, the estimate of cost is based only on information *within* an order. No attempt is made to gather information about the impact of an assignment from other orders. The other orders are potentially affected by an assignment via resource constraints and therefore represent untapped information.

A related technique for finding the impact of an assignment is the propagation of preferences through a temporal constraint graph [Sadeh 89] . All activities begin with a local preference curve over the values in its domain. The propagation, from an identified "central" activity, aggregates a curve reflecting the global preference for each possible reservation of that activity. The aggregation is done with a recursive algorithm resting on the original local preference curves. Propagation techniques are given for cyclic and acyclic temporal constraint graphs. Unfortunately, the complexity of the propagation scheme is very high and therefore the algo-

---

9. OR shadow prices reflect the impact on the total profit of an increase (or decrease) in one unit of resource, the MRP-STAR shadow prices reflect the impact on the total cost of changing the execution time of a job by one time unit.

rithms are intractable for even medium sized problems. Furthermore, the propagation is explicitly designed for temporal graphs and therefore is not easily applicable to gathering information from other orders.[10] We build on this work in the specification of techniques for the propagation of costs over all types of constraints.

2.6 Discussion

The work reviewed here falls into two broad categories:

- General techniques in the CSP literature.
- Specific techniques for areas such as scheduling and vision.

The general techniques typically address pieces of the relaxation problem but fail to represent key dynamic and semantic issues. The constraint optimization work, the PCSP work, and the selective non-satisfaction of constraints by static weights do not address the meaning of constraint relaxation. These techniques can not express that the impact of constraint relaxation depends highly on how the constraint is modified (e.g. *what* new value tuples are allowed). If constraint relaxation is to be used as an operator in search, we must have a more fine-grained control than simply ignoring or not ignoring a particular constraint. Because of the lack of semantics, it is hard to integrate meaningful heuristic decision making into these techniques. Without some basis on which to make search-guiding decisions, these exponential techniques can not be used on problems of a reasonable size.

We view the PCSP work as a special case of the general relaxation problem. If we define a relaxation problem such that all relaxation costs are identical it becomes a PCSP problem. In Chapter 4, we will compare the a number of relaxation algorithms to the PEFC3 algorithm on a number of PCSP problems.

In contrast, the more domain-specific literature allows the close integration of heuristics and begins to represent the meaning of constraint relaxation. The lookahead analysis done by the scheduling algorithms is designed to estimate the impact of a particular decision and in some cases (e.g. ISIS) the decision may concern how a constraint is to be relaxed. Only one of the methods for the estimation of the impact of a reservation can account for inter-order costs. The shadow price calculation [Morton 86] uses a iterative calculation to take into account both temporal and resource interactions. However, the authors note that the calculation can be expensive, the prices can be unstable, and that it is necessary to re-calculate the prices periodically. The other scheduling work, using a more explicit constraint model, does not address the propagation of information through all types of constraint. In Chapter 5, in order to redress

10. The authors address propagation in a temporal constraint graph with explicit disjunctions (e.g. activity A must occur before *or* after activity B). Resource constraints can be modeled with these constraints, however the complexity of propagation is even higher.

this non-generality, we propose techniques to estimate the impact of a activity reservation. This impact is expressed in terms of the minimum cost relaxations necessary if that value were to be put in place.

The repair-based scheduling algorithms embody the general concept of repair-based search around which it is possible to build a relaxation algorithm. In Chapter 4, we define a relaxation algorithm for PCSPs based on the MinConflicts scheduling algorithm. We compare this algorithm to other relaxation algorithms on the PCSP problems.

2.7 Summary

We have reviewed the previous work in a number of areas concentrating on work done in extending CSP techniques [Freuder 92] and work in scheduling [Sadeh 91] [Sadeh 94] [Minton 92] . We will revisit both of these areas in later chapters.

The previous work does not formulate a notion of constraint relaxation that is both expressive and flexible enough to be used in general constraint satisfaction problems while at the same time allowing the integration of heuristics to apply algorithms to specific problem domains.

We now present a propagation-based schema for constraint relaxation algorithms. Not only are constraints seen as a conduit for the transmission of information through the constraint graph, but they are in themselves loci of knowledge and computation. The work on preference propagation, reviewed above [Sadeh 89] , can be seen as a basis for the propagation scheme that we propose.

Chapter 3 A Schema for Constraint Relaxation

This chapter presents a general definition of constraint relaxation applicable across all constraint types. We introduce an algorithm schema that provides a framework for relaxation-based algorithms and the integration of heuristic techniques. It is necessary to take a general approach in order to make contributions toward constraint-directed reasoning theory as well as to ensure that our techniques will be applicable across a wide range of domains and to heterogeneous constraint models. Given the complexity of constraint-based problem solving, we require that domain specific heuristics can be easily integrated to the algorithmic schema.

We propose a general, propagation-centered schema for constraint relaxation. The constraint model is introduced and we elucidate the relaxation schema with an extended example. A more formal treatment of the basic propagation mechanism is then given. Finally, we suggest two approaches to dealing with the complexity of the relaxation schema.

3.1 Foundational Concepts

3.1.1 Constraint Model

We define the constraint relaxation problem as consisting of the following:

- V - a set of n **variables** $\{x_1, \ldots, x_n\}$ whose values come respectively from the discrete, finite domains $D_1, \ldots, D_n$.
- C - a set of m **constraints** $\{c_1, \ldots, c_m\}$. Each constraint, c_k:
  - is defined over a subset of variables, $\{x_i, \ldots, x_j\} \subseteq V$
  - contains a predicate, **Satisfied**$(x_i, \ldots, x_j)$, which returns TRUE to indicate that the constraint is satisfied by the current variable instantiation and FALSE otherwise. The predicate is defined on the Cartesian product of the domains of the relevant variables.
  - contains a function, **GenerateRelaxation**, which returns a set of constraints, RC_{c_k}, that are relaxations of the constraint. The set of value tuples that satisfy a relaxation is a superset of that which satisfies the original constraint. Relaxing a constraint is equivalent to replacing it with one of its relaxations.
  - contains a function, **RelaxationCost**(c_p), indicating the local cost incurred if the constraint is replaced by relaxation c_p. All costs are non-negative and the cost for a constraint to relax to itself is 0.
- $V_{c_i} \subseteq V$ the set of variables directly constrained by constraint c_i.

- $C_{v_j} \subseteq C$ the set of constraints on variable v_j.

A solution to a constraint relaxation problem is an assignment of a value to each variable (from its respective domain) such that all constraints or relaxations are satisfied and the total cost of the constraint graph (the sum of the cost of each relaxation) is as small as possible. A solution to the original problem, with no relaxation, has a cost of 0 and is considered optimal.[1]

The RelaxationCost at a constraint is well-defined but possibly expensive to calculate. We assume that modeling includes specification of a RelaxationCost function at each constraint.

Our model is a generalization of the usual CSP model and as such it can be used for CSP problem solving by ignoring the GenerateRelaxation and RelaxationCost functions. It also generalizes work done in areas related to constraint relaxation discussed in the previous chapter.

3.1.2 Search With Relaxation

In CSP search there is typically a single operator: the assignment of a value to a variable. Heuristic techniques are designed to limit the number of alternative values that must be explored. Backtracking occurs when it is found that a previous choice can not lead to a solution.

Relaxation-based search has two operators: the assignment of a value to a variable and the relaxation of a constraint. For any complete value assignment, we can always find a solution by relaxing constraints so that they are satisfied. Since we want to find a low cost solution, both the value assignments and the relaxations are important.

A generate-and-test algorithm can be used for relaxation problems. All possible value combinations are generated and, for each, the relaxations to find a solution are made. We store the solution and produce another valuation. If a subsequent valuation has a lower cost, the previously stored solution is replaced. For relaxation search, generate-and-test is more expensive than it is for CSP search because all possible value combinations must be produced.

3.2 Relaxation-Based Search

The proposed relaxation schema has two stages:

1. Value/control/cost propagation - a value is assigned and propagated through the graph. Control is then propagated to another node (constraint or variable) where further value and control propagation may be done. The value and control propagation eventually terminate and cost information, found on the basis of the value and control propagation, is gathered, cached, and propagated back. No commitment is made to particular assignments or relaxations.

1. In many cases there exists no optimal solution due to contradictory constraints.

2. Relaxation propagation - if the cost returned by the first stage is deemed acceptable, the graph configuration corresponding to that cost is put in place by propagation of relaxations. This propagation, depending on cost information cached in the first stage, makes commitments to specific variable instantiations and constraint relaxations.

In this section, we will define propagation and provide an extended example of a relaxation algorithm executed on a small constraint graph.

3.2.1 Propagation

Traditionally, propagation has been the transmission of information between variables over constraint links. We extend this definition by making a constraint a first-class object. We propose that a constraint is not simply a conduit of information but rather a locus of processing during the propagation procedure. The propagation is not simply *via* a constraint, rather information is transmitted from object to object, where the objects are variables and constraints. When propagation reaches an object, action is taken based on the received information.

We propose four types of propagation, as follows:

1. Value propagation - After a variable is assigned to a value (or after a constraint is relaxed), value propagation transmits the information effect to neighboring nodes. Typically, the effect of value propagation on a variable will be to prune the domain of possible values, because the decision invalidates some of the domain values. Similarly, at a constraint the number of possible relaxations will be pruned. Using the graph coloring example from Chapter 1, the assignment of blue to a variable followed by value propagation over a non-relaxable constraint, would prune the value blue from the adjacent variable.
2. Control propagation - When processing has completed at a node, the control proceeds to some neighboring node where further processing is done. Both the identity of the neighboring node and the processing undertaken upon arrival is dependent upon heuristics specified when the schema is instantiated. Usually, the action taken at a variable is the assignment of a value. At a constraint, the action is the selection of a constraint (a relaxation of the constraint or the constraint itself) that "replaces" the original constraint. The action may include value propagation or continuation of control propagation.
3. Cost propagation - Once the value/control propagation has terminated (see below for termination criteria), we propagate the cost associated with the graph back toward the variable where control propagation began (the *source variable*). This cost is a result of the scheduling decisions (the assignments and relaxations) made during the value/control propagation, therefore the cost information will flow backward along the path that control propagation originally took.
4. Relaxation propagation - The cost propagation provides aggregate information about the cost of the solution that can be found given a particular assignment of the source variable. On the basis of specified criteria, we may decide that the cost of one of the solutions is acceptable. Therefore, we commit to the value of the source variable corresponding to the solution. We perform a propagation that traces the original propagation in order to commit to assignments and relaxations searched through in the value/control/cost propagation

stage. Relaxation propagation consists of value propagation to promulgate the effects of a decision, followed by control propagation. In this case however, the action at each node will be to commit to a value assignment or relaxation based on cached cost information and commitments that have already been made.

Propagation is based on the connectivity of the constraint graph. Information from a variable can only be directly transmitted to a constraint on that variable. Similarly, the propagation from a constraint can only directly proceed to a variable attached to the constraint.

With perfect knowledge, a propagation-based algorithm is linear in the number of variables and constraints. At each variable the correct value is assigned and at each constraint the optimal relaxation is made. We do not have this knowledge, therefore it is necessary to try different values at a variable and different relaxations at a constraint. The set of values that we search over at a variable is called the *candidate value* set. The set of relaxations that we attempt at a constraint is the set of *candidate constraints*.

For the balance of this chapter, we refer to the value and control propagation as value propagation. This simplification is made because we assume (in this chapter) that the effects of a decision only flow directly to the node to which control flows. It is not the case that the effects of an assignment flow to all neighboring nodes and then control flows to only one. Both the effects of an assignment and control flow to one neighboring node. Therefore, value propagation will include transmitting the information on an assignment and the control to a unique neighboring node. In Chapter 5, we relax this simplification and implement separate value and control propagation.

3.2.2 An Extended Example

Assume we choose x_1 as the starting point in the graph shown in Figure 5. At each variable, the candidate value set will contain all values that are consistent with previously made relaxations. Similarly, at each constraint the candidate constraint set contains all constraints that are consistent with previous value assignments.

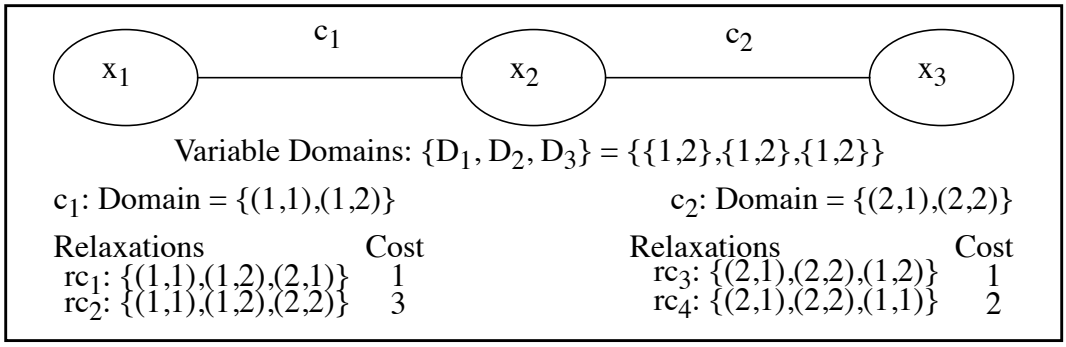

Figure 5. A Simple Constraint Graph

The following is a trace of the search of the graph in Figure 5:

1. Select the candidate values set at x_1. In this case, we choose all domain values as candidates; therefore the candidate value set is {1, 2}. Choose one of the candidate values and temporarily assign x_1 to that value. Select $x_1 = 2$.
2. Value propagate to *c1*. The candidate constraint set is chosen such that all elements are consistent with the previous assignment, $x_1 = 2$. The constraint itself, c_1, is not consistent. The candidate set at *c1* is $\{rc_1, rc_2\}$. Choose one of these elements. Relax c_1 to rc_1 at a cost of 1.
3. Value propagate to x_2. The choice of the relaxation at c_1 places a constraint on the possible candidate values at x_2. In fact, the candidate set is a singleton: {1}. Assign $x_2 = 1$.
4. Value propagate to c_2 and find the local candidate constraint set. Based on $x_2 = 1$, the set of candidate constraints is $\{rc_3, rc_4\}$. Pick rc_3 and relax c_2 at a cost of 1.
5. Value propagate to x_3. Based on the decision at c_2, the only possible value for x_3 is 2. There are no further constraints, therefore, cost propagation begins. The assignment $x_3 = 2$ does not, in itself, incur any cost, therefore a cost of 0 is propagated back to c_2.
6. Cost propagate to c_2. The local cost of relaxing to rc_3 is added to the cost returned from x_3. Store the tuple $(x_2 = 1, rc_3, 1)$. This tuple indicates that, so far, the lowest cost for the part of the graph to the right of c_2, when $x_2 = 1$ is 1. This cost is found by relaxing c_2 to rc_3. There remains a candidate constraint that has not been evaluated, so relax c_2 to rc_4 at a cost of 2.

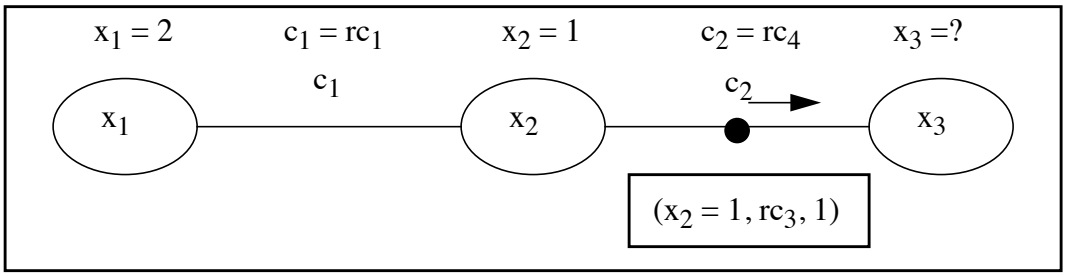

Figure 6. Snapshot of the Graph After Step 6

Figure 6 presents the graph after step 6. The black dot on c_2 indicates the position of the search while the arrow indicates that value propagation is about to proceed to x_3. All the assignments shown above the variables and constraints are temporary. Since the search has entered x_3 and left again, the value of x_3 is undefined at this point.

7. Value propagate to x_3 and assign $x_3 = 1$. The same action will take place as in the previous value propagation to x_3. It is recognized that no further constraints exist, so a cost of 0 is propagated back to c_2.
8. Cost propagate back to c_2. We add the propagated cost to the local cost and form the entry $(x2 = 1, rc_4, 2)$. This cost is greater than the cost of the other candidate constraint (see step 6), therefore the new tuple is not saved. All possible candidate constraints have been tried, so a cost of 1 (the lowest cost solution we have found) is propagated to x_2. If there were additional candidate constraints, steps 7 and 8 would be repeated for each one.

9. Cost propagate to x_2. Form the cost entry ($x_2 = 1, 1$). Because there are no other cost entries with the index $x_2 = 1$, this one is stored, indicating that it is the lowest cost solution we have found. There are no other candidate values for x_2 (see step 3, where the candidate value set was created), therefore cost propagation continues.
10. Cost propagate to c_1 and add the local relaxation cost to the propagated cost. Form and save the cost entry: ($x_1 = 2, rc_1, 2$). This indicates that when $x_1 = 2$, the lowest cost partial solution we have found is 2. This cost corresponds to relaxing c_1 to rc_1. Not all candidate constraints have been tried, so we relax c_1 to rc_2 at a cost of 3. Value propagation begins again to assess the cost of the subgraph with the new assignment.
11. Value propagate to x_2. The relaxation rc_2 constrains the value of x_2 to be 2. This is the only member of the candidate value set, so assign it to x_2.
12. Value propagate to c_2. All local constraints are consistent with $x_2 = 2$, therefore the candidate constraint set is $\{c_2, rc_3, rc_4\}$. Each of these constraints is assessed in turn just as was done above in steps 4 through 8. The ultimate local result is a cost entry ($x_2 = 2, c_2, 0$), indicating that when $x_2 = 2$, the lowest cost solution to the subgraph is 0 and this occurs when c_2 is not relaxed. Propagate a cost of 0 back to x_2.
13. Cost propagate to x_2 and store ($x_2 = 2, 0$). There are no more candidate values so cost propagation continues.
14. Cost propagate to c_1. A cost of 0 is propagated back to c_1 and this is added to the local cost to form the entry ($x_1 = 2, rc_2, 3$). The cost of 3 is higher than the previous entry ($x_1 = 2, rc_1$, 2) (see step 10) so the newer entry is discarded.

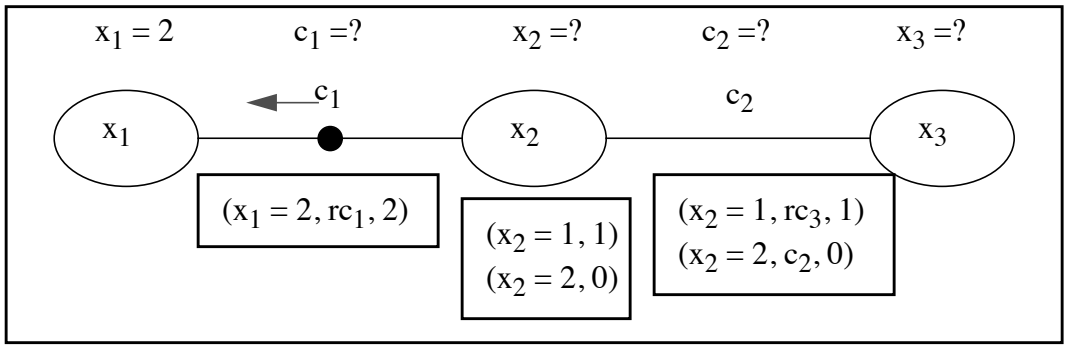

Figure 7. Snapshot of the Graph After Step 14

Figure 7 shows the search after step 14. Again the black dot indicates the current location of the search. The arrow indicates that cost propagation is about to proceed to x_1. As with Figure 6, when cost propagation leaves an object, the value (or relaxation) is undefined.

15. Cost propagate a cost of 2 to x_1. Store the cost entry ($x_1 = 2, 2$). There is another candidate value for x_1. Temporarily assign $x_1 = 1$ and value propagate to c_1. Steps 1 through 14 are repeated for the new value of x_1.

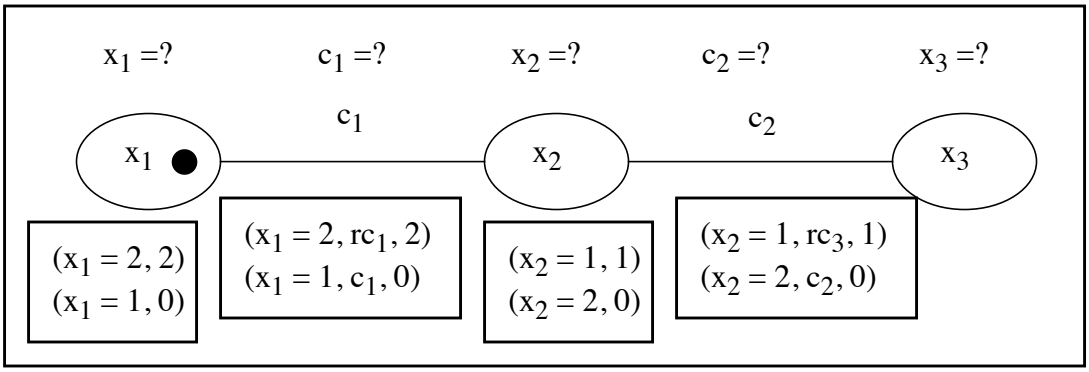

Figure 8. The Graph after All Value/Cost Propagation

Figure 8 displays the cost information that is gathered in the value/cost propagation phase. The information at x_1 indicates that an assignment of $x_1 = 2$ results in a solution of cost 2, while $x_1 = 1$ leads to a zero-cost solution. The information stored at the other objects allows either solution to be re-created. The actual modification of the graph occurs in the relaxation propagation phase of the relaxation algorithm. The first step is to choose a value for the source variable based on some criteria. Typically, we choose the value that leads to the minimum global cost solution. For the example, however, we choose $x_1 = 2$ because it requires some relaxation to be done.

The relaxation propagation takes the following steps (having chosen and assigned $x_1 = 2$):

1. Propagate relaxation to c_1. At c_1, use the assignment $x_1 = 2$ to find the correct relaxation in the cache. It is the first entry shown in Figure 8: c_1 is relaxed to rc_1.
2. Propagate relaxation to x_2. The relaxation rc_1 requires $x_2 = 1$. Assign this value. Had the relaxation of c_1 allowed a number of assignments of x_2, the lowest cost assignment from the cost cache would be chosen.
3. Propagate relaxation to c_2. Locate the entry ($x_2 = 1$, rc_3, 1) which is the only entry with $x_2 = 1$. Relax c_2 to rc_3.
4. Propagate relaxation to x_3. The relaxation of c_2 to rc_3 together with the assignment $x_2 = 1$, requires $x_3 = 2$. Assign this value.

The solution when $x_1 = 2$ is shown in Figure 9.

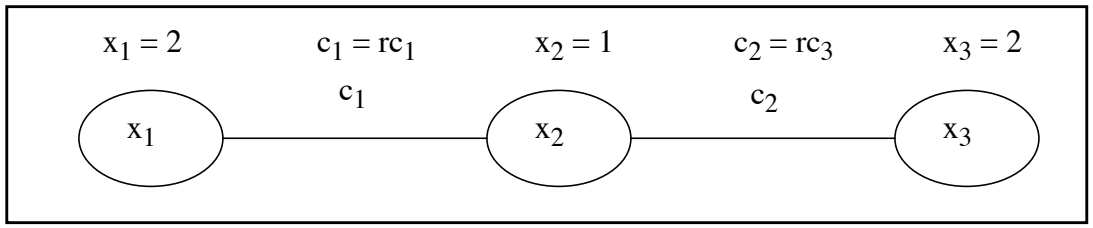

Figure 9. The Graph of the Final Solution for $x_1 = 2$

3.2.3 Comments on the Example

There are a number of points that the extended example raises.

- The search is done in the value/cost propagation phase. The relaxation propagation depends on the stored cost information.
- The lowest cost partial solution for each candidate value is stored in the cost cache at each variable. The cache is guaranteed to be no larger than the domain of the variable.
- There is an entry in the cost cache at each constraint for every candidate value of the previous variable. This cache can be no larger than the domain of the previous variable.
- The information stored by the value/cost propagation is directional. The relaxation propagation must flow along each constraint in the same direction as the value propagation.
- If a variable has more than one constraint, the costs returned from each constraints are summed. If x_2 was the starting variable, the value propagation would proceed along c_1 and c_2 independently. The costs returned must be summed to form the cost entry at x_2.
- All assignments made during value/cost propagation are temporary. This does not prevent the variable from already having a value. In a repair-based application, we can use the current value to aid in the selection of the candidate values and candidate constraints.
- If the size of the candidate constraint set at each constraint is k, and if p values are chosen at each variable, then the complexity is $O(p^n k^m)$, where m is the number of constraints and n is the number of variables reached in the propagation.

3.3 The Relaxation Schema

The relaxation schema is based on the propagation of information through the constraint graph. The propagation is modulated by procedures that define the behaviour at five *heuristic decision points*. Given the exponential complexity of the search, if a relaxation algorithm is to be of practical use, we need to use heuristics to limit the selection of candidate values and constraints. The decision points are the key places in the algorithm where heuristics can have significant impact on the solution quality vs. search effort trade-off. By selecting a small set of alternatives from the possibilities, the complexity of the search will be reduced to a manageable level, however the number of solutions investigated will be lessened. The selection of small sets will result in a fast algorithm but a sub-optimal solution.[2]

The heuristic decision points are:

1. Selection of the *source variable*.
2. Selection of a set of *candidate values* for each variable.
3. Selection of a set of *outgoing constraints* from each variable.
4. Selection of a set of *candidate constraints* at each outgoing constraint.

---

2. Despite the isolation of these decision points, there is no requirement to actually use heuristics in an instantiation of the schema. An algorithm can be defined which exhaustively searches through all the possibilities at each decision point.

5. Criteria for selection of a *value-commitment* at the source variable.

The *source variable* is the origin of the value propagation. Given the effects seen for different variable orderings in CSP solving, we expect this choice to have an impact on performance.

A set of *candidate values* are chosen for the source variable and for each variable that is entered by value propagation. It is not necessary that the same selection method be used for the candidate value set at each variable.

The *outgoing constraints* are those along which propagation proceeds.[3] At all variables except the source variable, the selection is required to choose a subset of the constraints on the variable.[4] The set of outgoing constraints defines the subgraph over which alternatives are assessed. As with candidate value selection, the same method need not be applied at every variable. Typically, selection of outgoing constraints will be based on the properties of the constraint itself (e.g. its type or if it is relaxable) or on properties of subgraphs in which the constraint participates (e.g. complete subgraphs).

Each candidate value is propagated along each outgoing constraint. *Candidate constraints* are selected at each outgoing constraint. A candidate constraint is either the constraint itself or a relaxation of the constraint produced by the **GenerateRelaxation** function.

After, the value/cost propagation has returned to the source variable, it is necessary to choose a value to assign to the source variable. This *value-commitment* is the beginning of the relaxation propagation and will be based on the gathered cost information. Typically, the value chosen will be the one that leads to the lowest global cost, however it is not necessary to commit to a value. Because the heuristic decision points specify that only a subset of possible solutions are searched over, it may be the case that the none of the solutions meet the criteria. In this case, no relaxation propagation is done, the gathered costs are discarded, and another source variable may be chosen. By making the value-commitment a heuristic decision point, we are allowing the integration of such techniques as simulated annealing [Kirkpatrick 83] , where an new solution is entered based on a complex function of the its cost and the amount of processing already done.

The relaxation algorithm used in the example is specified as follows:

- Source variable: x_1.
- Candidate values: All values in the domain of the variable that are consistent with previous choices.
- Outgoing constraints: All constraints except the constraint from where the value propagation entered a variable. For example, c_1 is not an outgoing constraint at x_2 because value propagation visited c_1 before reaching x_2.
- Candidate constraints: All relaxations that are satisfied by the previous decisions.

---

3. Because we are combining value and control propagation, we specify a single set of outgoing constraints. More generally, there is a set of outgoing constraints for value propagation and a set for control propagation.

4. A constraint can only be an outgoing constraint for one variable.

- Value-commitment: Arbitrarily choose $x_1 = 2$, as it requires relaxation of some constraints.

It is useful to re-iterate that the selection of the sets of candidate values, outgoing constraints, and candidate constraints limits the size of the graph and the size of the search on the subgraph. If the heuristics are based on local data or data gathered from a subgraph (e.g. local relaxation costs), the relaxation algorithm may be incomplete: it is not guaranteed to find the minimal cost solution. We see this as a classic trade-off between processing time and the quality of solution. An area for future work is the characterization of the types of trade-offs made by relaxation algorithms using different heuristics to limit the search (see Chapter 7).

3.4 Basic Propagation

With the intuition provided by the extended example, we now examine more formally the basic propagation mechanisms upon which the schema is built. These propagation methods build on work on preference propagation [Sadeh 89] .

3.4.1 Value and Cost Propagation

Before propagation can begin, a variable, $x_{source} \in V$, a set of candidate values for that variable, $VAL_{source} \subseteq D_{source}$, and a set of outgoing constraints, $CON_{x_{source}}$, must be selected. We will assume these tasks have already been done. For each value, $v_j \in VAL_{source}$, we attempt to answer the question: what is the minimum cost graph we can find with $x_{source} = v_j$? To answer this question, we calculate the *PostCost*: v_j is propagated along each outgoing constraint and the costs that are returned are summed.

$$PostCost(x_{source} = v_j) = \sum_{c_i \in CON_{x_{source}}} ConCost(c_i | x_{source} = v_j) \quad \textbf{(1)}$$

The *ConCost* function can be broken into two other functions.

$$ConCost(c_i | x_a = v_j) =$$

$$min_{rc_k \in CC_{c_i}} (RelaxationCost(rc_k) + PostCost_{rc_k}(x_b = v_p)) \quad \textbf{(2)}$$

Where:

c_i is the constraint connecting x_a and x_b.
CC_{c_i} is the set of candidate constraints for constraint c_i.
RelaxationCost(rc_k) is the RelaxationCost function of c_i.
v_p is a value (in the domain of x_b) such that **Satisfied**(rc_k | v_j, v_p) is TRUE.

The subscript on the *PostCost* function indicates the constraint that is traversed to reach the variable. The subscripted constraint can not be an outgoing constraint by the *PostCost* function.

At a variable, the cost for the each candidate value is assessed by calculating the *PostCost*. The (value, cost) tuple corresponding to the lowest cost is stored at the variable. When all candidate values have been assessed, the cost element of the stored tuple is propagated along the constraint by which value propagation originally reached the variable.

At a constraint, the cost for the first candidate constraint is found using Equation (2). The (value, relaxation, cost) tuple for the lowest cost relaxation is stored. This tuple represents the value of the previous variable, the lowest cost relaxation that can be made, and the cost of that relaxation. Note that the cost here is not simply the local RelaxationCost, but rather the Con-Cost. When all candidate constraints have been assessed, the lowest cost is propagated to the variable from where value propagation originally reached the constraint.

3.4.2 Relaxation Propagation

Relaxation propagation is the commitment stage of the relaxation schema. Based on the information gathered in the value/cost propagation stage, minimum cost values are assigned to variables and constraints are replaced by relaxations. The relaxation propagation begins by assigning one of the candidate values to the source variable and propagating along each outgoing constraint. At each constraint, the value is used to locate the tuple containing previously assessed lowest-cost candidate constraint. This relaxation is put in place and propagation continues to the next variable. At each variable, the incoming information is used to locate the appropriate local value. As in the value propagation, this process stops when a variable with no outgoing constraints is reached. Pseudocode for relaxation propagation is in Figure 10.

```
assign lowest cost candidate value to the source variable
propagate to all outgoing constraints

for each constraint entered{
   use the value to look-up the lowest cost entry in the cost cache
   replace the constraint by the relaxation in the cost cache entry
   propagate the relaxation to the adjacent variable
}

for each variable entered{
   use the propagated information to look up the lowest cost
      consistent value in the cost cache
   assign the value to the variable
   propagate value to all outgoing constraints
}
```

Figure 10. Pseudocode for Relaxation Propagation

No search is involved in relaxation propagation. The value that is propagated from a variable uniquely identifies the appropriate candidate constraint value. Each constraint is visited exactly once during the relaxation propagation.

3.4.3 Termination of Propagation

It remains to be shown that propagation will terminate on any constraint graph. The cost and relaxation propagation only retrace constraints and variables visited by value propagation.[5] Therefore, it is sufficient to show that the value propagation will terminate.

First, note that the sets selected at each decision point are finite. The candidate value set is finite because it is a subset of the domain of the variable and, by definition, the domain is finite. The set of candidate variables at a constraint is a subset of all possible relaxations of the constraint plus the constraint itself. A relaxation must be satisfied by at least one element in the Cartesian product of the domains of the variables which it constrains. Therefore, the maximum number of relaxations of a constraint is equal to the size of the power set of the Cartesian product of the relevant variable domains. Since both the number of variables and the size of the variable domains is finite, the number of relaxations must also be finite. Finally, the total number of constraints in a graph is finite, therefore the constraints on one variable must be finite, and it follows that the set of outgoing constraints at a variable is finite.

Value propagation terminates when a variable selects an empty set of outgoing constraints. We specify that the set of outgoing constraints can not contain the constraint that was propagated along to reach the variable. In an acyclic graph, we are guaranteed to reach a variable that has a single constraint. The single constraint is where that value propagation entered the variable, therefore the variable must select an empty set of outgoing constraints. In this case we define the *PostCost* function at that variable to be equal to 0.

Given a cyclic graph, it is possible for value propagation to leave a variable and re-enter it via another constraint. To ensure termination, we expand the termination criteria for value propagation. In addition to stopping when a variable selects an empty set of outgoing constraints, value propagation must also stop when it re-enters a variable along a different constraint than previously. To detect this occurrence, two registers are created at each variable. The first contains the **AlreadyVisited** flag that is set when a value is propagated to the variable and unset when cost propagation leaves the variable. The second register, $v_{already}$ stores the currently assigned candidate value. If the variable is re-entered during value propagation, the cycle is detected because the **AlreadyVisited** flag is already set. We redefine *PostCost* as in Figure 11.

5. Cost propagation traces the value propagation in the opposite direction.

```
if AlreadyVisited is set AND v_already is consistent with
      previous relaxation
   PostCost = 0
else if AlreadyVisited is set AND v_already is inconsistent with
      the previous relaxation
   PostCost = ∞
else use Equation (1)
```

Figure 11. Redefinition of the PostCost Function to account for Graph Cycles

In normal value propagation, the choice of a candidate value at a variable is made so that the values are consistent with the previous relaxations. In this case however, we already have a value for the variable. If the value happens to satisfy the relaxation, a cost of 0 is propagated. Conversely, if the value does not satisfy the constraint, a cost of ∞ is propagated. In the latter situation, the relaxation is attempting to constrain the variable to a value different from the one to which it is assigned. In other words, the variable is constrained to be two values at the same time. This is clearly impossible. The effect of returning a cost of ∞, will be to remove this partial solution since any alternative with a cost < ∞ is rated more highly.

This re-definition of PostCost guarantees termination of value propagation on graphs with cycles. Whenever a variable is re-entered, cost propagation immediately begins.

3.5 Relaxation in Non-Trivial Graphs

Two problems occur when a naïve relaxation algorithm is executed on a large graph that contains cycles: the exponential complexity of the algorithm and incorrect cost information in graphs with cycles. In any non-trivial problem, both of these difficulties must be addressed. We propose two solutions:

1. Cache more information at each variable and constraint. We build on the caches used for cycle detection by storing search-specific information at variables and constraints. This information is used to eliminate search repetition and provide accurate cost information. The former is achieved by recognizing when it is possible to cut-off value propagation because previous information applies to the new situation. The latter point applies to cyclic graphs where the existence of a cycle can lead to counting some constraint costs more than once during value/cost propagation. We can also bound the search by the minimum cost solution found so far. These methods, described fully in Chapter 4, will not deal well with the time complexity: the worst case time complexity is still exponential. In some sense, this method trades time complexity for space complexity. By caching information, we avoid search but the space complexity is high.
2. Use texture measurements to create abstractions and find tractable subgraphs. Recall that a texture measurement assesses the extent to which a particular graph or subgraph has some property. For example, the algorithms presented in Chapter 5, use texture measurements to

identify a subgraph of tasks that form a good estimate of the global impact of scheduling decisions. By the exploitation of the problem properties, the number of variables and constraints over which propagation is done can be strictly limited. Relaxing different, small subgraphs (e.g. 5 to 8 variables) and choosing very few candidate values and candidate constraints will minimize complexity difficulties. In dealing with graphs with cycles, the subgraphs can be specifically chosen to be acyclic or techniques from the above approach can be applied.

We view the second approach as more general and likely to result in usable algorithms. It concentrates the relaxation algorithm on small subgraphs that are independent of the total graph size. Despite the complexity difficulties with the first approach, these techniques are necessary if the subgraphs found by texture measurements contain cycles. This approach is applicable to small problems (i.e. about 20 variables) such as Partial Constraint Satisfaction Problems (PCSPs) [Freuder 92] .

3.6 Summary

This chapter has presented a relaxation schema based on propagation of information through constraint graphs. It specifies five decision points that define the amount of propagation done in an instantiation of the schema. Given that the complexity of the search is exponential, we must develop ways to efficiently identify good relaxations in non-trivial graphs. We sketch two approaches to this problem. The first utilizes caches of information at each variable and constraint in order to identify cycles in the graph and eliminate repetition of search. The second uses texture measurements to identify important subgraphs and graph properties that allow use of relaxation algorithms where the propagation is restricted.

In the following chapters we present algorithms defined using each of these approaches. In Chapter 4, we examine the difficulties with cyclic graphs and employ information caches at each variable and constraint to deal with these difficulties. Three repair-based algorithms for PCSPs are defined and their performance is compared with the PEFC3 algorithm. In Chapter 5, we present a number of relaxation algorithms that use texture measurements to estimate the impact of scheduling decisions in schedule optimization problems. We compare the estimates to a baseline estimate found by Monte Carlo simulation.

Chapter 4 Cache-Based Constraint Relaxation

This chapter investigates the use of information caches at each variable and constraint to minimize search and ensure that the cost incurred by a cycle is correctly calculated. We define three repair algorithms using the cache techniques and apply them to partial constraint satisfaction. We present an empirical evaluation of the performance of a number of variations of these three algorithms to PEFC3.

4.1 Difficulties With Cycles

When a graph contains cycles, an algorithm defined within the relaxation schema of the previous chapter will encounter two difficulties:

1. The cost of a cyclic subgraph will be counted more than once.
2. The information necessary for relaxation propagation may be corrupted.

We will examine each of these more closely and propose solutions.

4.1.1 Counting Cycle Costs

We use Figure 12 to demonstrate the first difficulty. Figure 13 shows a close-up of the propagation surrounding variable x_1.

Value propagation enters x_1 along c_0. Constraints c_1 and c_2 are selected as outgoing constraints and the propagation proceeds along c_1 (Figure 13A). Value propagation will re-enter x_1 via constraint c_2. The cycle is detected and cost propagation immediately begins back along c_2 (Figure 13B). Cost propagation eventually returns to x_1. An instantiation of the schema in the previous chapter would then propagate the same candidate value along c_2 and, when a cost is returned, sum the two costs. The subgraph assessed by propagation along c_1 is exactly that assessed by propagation along c_2. We count the cost of the subgraph twice.

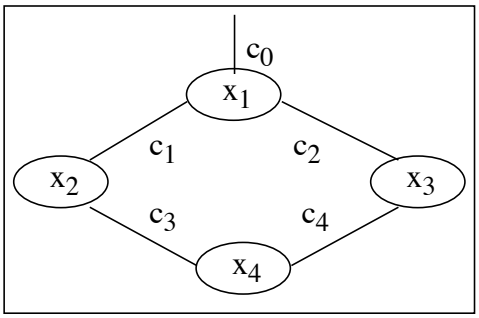

Figure 12. A Constraint Graph with a Cycle

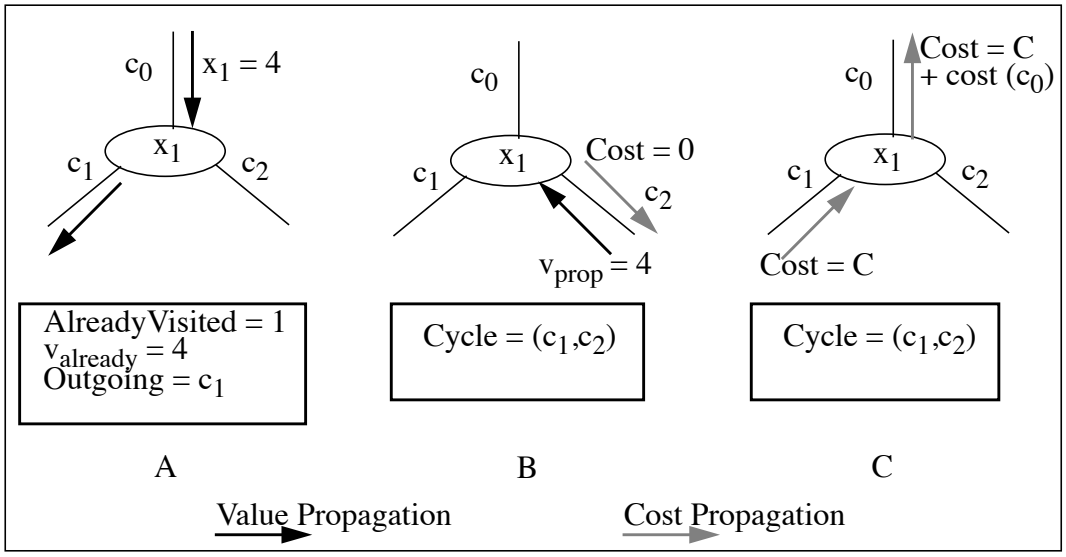

Figure 13. A Close-up of the Constraint Graph in Figure 12

Since the value propagation from x_1 along c_2 covers exactly the same constraints covered in the previous propagation (along c_1), we need not value propagate along c_2. The next candidate value for x_1 is selected or, if no candidate values remain, cost propagation continues back along c_0 (Figure 13C).

To detect when value propagation should not be done, we store additional information at each variable to identify co-cyclic constraints. As shown in Figure 13A, the identifier of the outgoing constraint is recorded: constraint c_1 at variable x_1. Upon re-entering a variable, we note that the recorded constraint and the constraint that was returned on (c_2 in this case) are co-cyclic. On returning the cost to x_1, we know from the stored cycle information that we should not follow the usual form of Equation 2 (see Chapter 3, Section 3.4.1). The set of outgoing constraints is dynamically reconfigured by removing any constraint that takes part in a cycle. Our knowledge of co-cyclic constraints comes from re-entry into a variable after value propa-

gation along one constraint in the cycle; therefore we are guaranteed to propagate along only one co-cyclic constraint in each cycle. This technique generalizes to find multiple constraints at a single variable that are part of the same cycle.[1] In Figure 14, if x_1 is the initial variable, c_1, c_2, c_5, and c_6 are all part of cycles formed around x_1. We record all the co-cyclic constraints at the variable in an unsorted list.

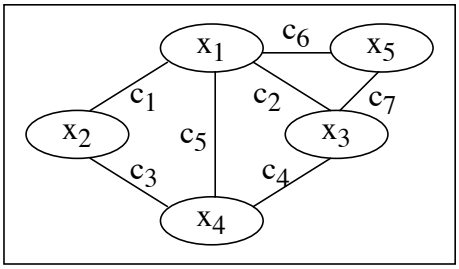

Figure 14. A Constraint Graph with Cycles

A drawback of this mechanism is that solutions may be overlooked. In Figure 14, if value propagation begins at x_1 and proceeds along c_2, the set of candidate constraints at constraint c_4 will be selected when propagation reaches the constraint from x_3. If value propagation had proceeded along c_1 or c_5, the candidate constraints at c_4 would have been selected during propagation from x_4. Depending on the heuristic used as a basis for this selection, it is possible that the two sets would be different and one might lead to a better solution. Because we only propagate from x_1 along one of c_1, c_2, or c_5, we only generate the one of these candidate sets.

For example, c_4 is defined as shown in Figure 15. In picking the set of candidate constraints we simply pick the lowest cost, consistent constraint (or relaxation). Suppose value propagation flows along c_3 to x_4 and constrains x_4 to be 1. The candidate constraint for c_4 will be the constraint itself, since it is consistent with $x_4 = 1$. However, if the value propagation flows along c_2 (or c_7) to x_3 and constrains x_3 to be 3, then the rc_1 is the candidate constraint at c_4. Unless the search backtracks and propagates a new value to x_3 (allowing $x_3 = 2$), the local, zero-cost solution will not be found when the value propagation flows along c_2 or c_7 to x_3.

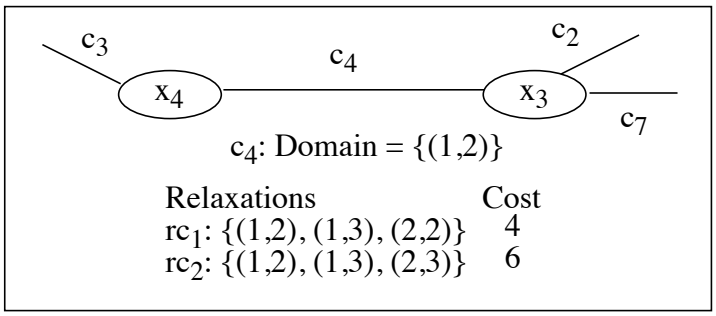

Figure 15. A Close-Up of a Subgraph from the Graph in Figure 14

1. More accurately, any pair of outgoing constraints in the cycle list are part of a cycle. Multiple cycles can exist.

Given that the algorithms within the schema will attempt to limit the number of alternatives that are explored, the increase in complexity necessary to avoid this problem is not warranted. We accept that in relaxation on cyclic graphs, we may not be able to try all alternatives.

4.1.2 Relaxation Information

Relaxation propagation depends on the (value, relaxation, cost) tuple recorded at each constraint. Recall that the value is a candidate value of the previous variable and that it is used as an index to the cost information during relaxation propagation. Because the value of the previous variable is used as an index, relaxation propagation must enter a variable from the same constraint as in value propagation. If it does not, then the "previous" variable is not the same variable as in value propagation, and the cost cache is meaningless.

We can ensure correct direction of relaxation propagation in a cyclic graph by modifying the cycle recording mechanism described above. Instead of using an unsorted list to store co-cyclic constraints, the outgoing constraint along which value propagation was done is the first element in the cycle list. Relaxation propagation can now identify the correct constraint to traverse. The other constraints in the list will be relaxed when they are reached later in the relaxation propagation.

A more serious problem concerns the cost cache at each constraint and variable. Recall that the cache at a constraint is indexed by a candidate value of the previous variable. The same value can be propagated to a variable more than once, resulting in the possibility that cache elements may be overwritten. Without cycles, this is not a problem because the cost of a subtree is the same whenever an identical value is propagated to its root. When cycles are allowed, the "subtree" stemming from a variable, x_b, can contain a variable, x_a, that has already been given a value. When the next value is propagated to x_b, the value of x_a may have changed (due to cost propagation back to x_a and the choice of another candidate value). This changed value can have significant impact on the cost of the "subtree" at x_b. If we index by the candidate value of the previous variable, we will overwrite the cache with new, *different* entries. Relaxation propagation may require one of the older entries that no longer exists.

Figure 16A shows the value propagation entering the subgraph along c_0 and proceeding along c_1. Eventually, the value propagation ends when x_1 is re-entered from c_3. Because the values $x_3 = 1, x_1 = 1$ are consistent with constraint c_3, the entry $(x_3 = 1, c_3, 0)$ is stored at c_3. Cost propagation returns through x_1 and eventually, due to a decision further back in the graph, a value of $x_1 = 2$ is propagated (Figure 16B). Again, value propagation continues until x_1 is re-entered from c_3. In this case, however, the constraint c_3 must be relaxed to rc_1 in order to allow the values $x_3 = 1, x_1 = 2$ to be consistent. Therefore, the cost cache entry $(x_3 = 1, rc_1, 3)$ is stored at c_3. Because the cost cache is indexed by the candidate value, the previous entry is overwritten. The information on how to correctly relax c_3, if $x_1 = 1$ is assigned during relaxation propagation, is no longer in the cost cache.

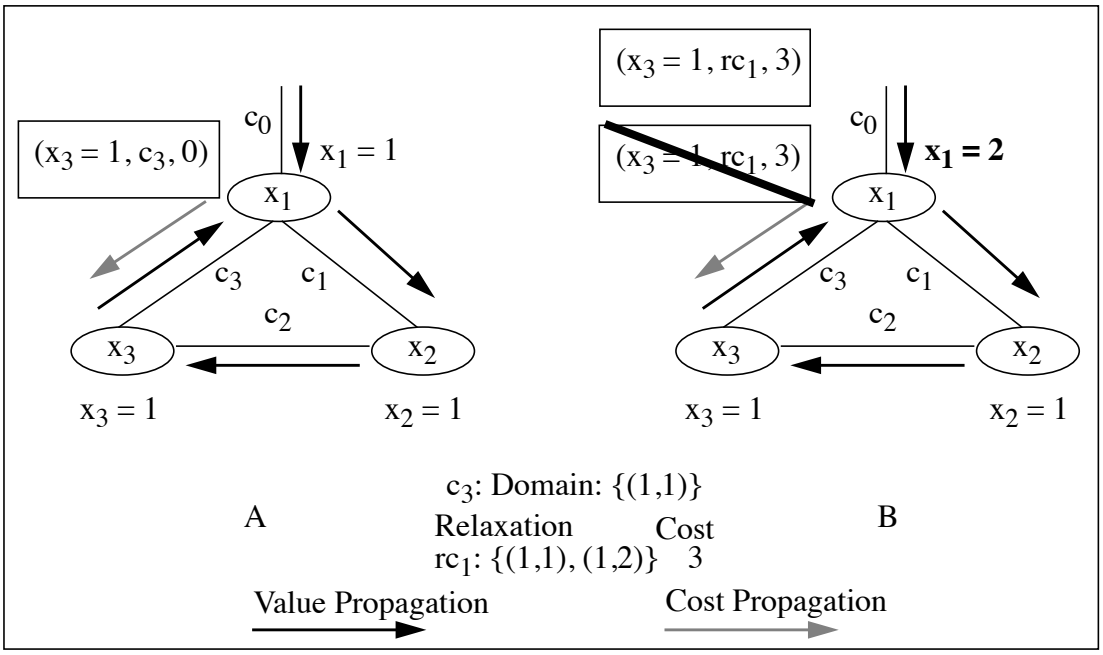

Figure 16. Overwriting the Cost Cache at a Constraint

A solution is achieved by attaching a *context* to the stored candidate constraint. The problem manifests itself when a cycle causes the value propagation to return to a variable. When a cycle is detected, we attach the (variable, $v_{already}$) pair to the cost that is propagated back. This context is stored with the tuple at each constraint. The relaxation propagation mechanism now not only uses the propagated value as an index, but also ensures that the stored context matches the current context. The match is found by checking that the stored value of each variable in the context corresponds to the value that the variable has been assigned.

In terms of the example in Figure 16, the original cost cache entry at c_3 is now ($x_3 = 1, c_3, 0$, (x_1, 1)). The subsequent cost cache entry is ($x_3 = 1, rc_1, 3, (x_1, 2)$). The second entry does not overwrite the first because the contexts are not equal. Relaxation propagation will therefore be able to locate the correct cost cache entry when it is making a commitment to a solution.

Since there can exist multiple cycles in the graph and many variables can appear “later” in the search tree, we must use a list of (variable, $v_{already}$) pairs for the context. When the cost returned by each constraint to a variable is summed, the union of the attached contexts is taken. The summed cost and the united context are then propagated. No entries are removed from the context until a variable appearing in the context is reached in cost propagation. At that point, the variable removes its own (variable, $v_{already}$) pair from the context list. The element can be deleted because all variables higher in the search tree were given a value before this variable, and therefore do not depend on the variable having a particular value.

4.1.3 Complexity

The two techniques above make use of caches at each variable and constraint. Lists of co-cyclic outgoing constraints are used to ensure proper cost counting of cycles. The total number of all elements across all lists at a variable is limited to the number of constraints on the variable. Each constraint only appears in one list because the lists represent independent cycles stemming from a variable. If a constraint were on more than one list, the cycles represented by the two lists would not be independent.

Use of the context mechanism invalidates the space-complexity argument for the cost caches given in the previous chapter. There we showed that the total size of the cache at a constraint was limited to the number of candidate values at the previous variable. The context mechanism uses an entry at a constraint for each value of the previous variable and for each value of the variables in the context. Therefore, the space complexity for a cost cache at a single constraint is $O(Dk + 1)$, where D is the maximum domain size and k is the maximum number of variables that can be in a single context. The value of k will depend on the structure of the graph. The worst case complexity occurs when the final variable to be visited in value propagation shares constraints with all other variables. The cache size at the constraint that was traversed to reach this final variable will have a cache size of $O(Dn - 1)$, where n is the total number of variables in the problem. Each constraint has a cost cache, therefore the complexity is further multiplied by the number of constraints.

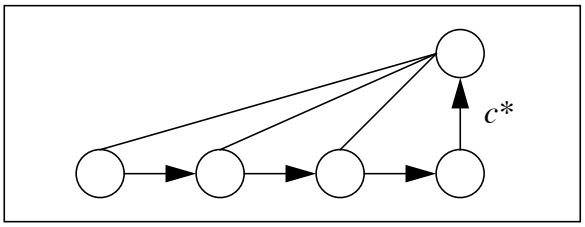

Figure 17. A Graph Corresponding to the Worst-Case Space Complexity of the Context Mechanism

Figure 17 demonstrates the worst-case situation on a small graph. The constraints with arrows represent the path that value propagation traveled to the final variable. The cost cache at the c* constraint will contain $O(D4)$ entries.

4.2 Using Information Caches to Limit Search

We can use the cost caches at each variable and constraint to limit search by eliminating repetition and by cutting off the search when it is found that any solution to a subgraph will have a larger cost than a previously found solution.

4.2.1 Using Stored Costs

In an acyclic graph, the minimum cost of a subtree with a particular value at the root will not change when a new value is given to a variable higher up the tree. Significant time savings can be achieved by storing the cost of the subtree for each candidate value of the root of the subtree. When value propagation returns to a variable (after cost propagation has left), the new set of candidate values is created. If this set contains elements also in previous candidate sets for this variable, all that is necessary is a look-up of the previously calculated cost. Value propagation for a repeated value is unnecessary because all information stored in the subtree the last time the value was propagated is still valid.

When cycles exist, the caching mechanism must record the context. The argument used above to show the necessity of the context mechanism for relaxation propagation applies to caches at variables. In order to cut off value propagation for a particular candidate value, we must find the value in the cache and match the stored context with the current situation. By adding contexts to the caches at each variable, we again increase the space complexity of the context mechanism. Overall the worst-case space complexity of the caches at each constraint and variable is $O(nmD^{n-1})$. Where n is the number of variables, m, the number of constraints, and D, the maximum domain size of any variable. This result strongly argues that the identification of subgraphs via texture measurements is necessary for practical application of these algorithms.

4.2.2 Bounding the Search

Work can be eliminated by ending part of the search when it is clear that any solution will not have a cost lower than that the current best cost. We propagate the current lowest cost along with the value propagation and filter this bound at each variable and constraint.

```
current bound = bound passed to variable
for each candidate value{
   for each outgoing constraint{
      get cost by propagation along outgoing constraint with
         current bound
      current bound -= cost
      total cost += cost
   }
   if total cost < current bound
         current bound = total cost
}
```

Figure 18. Filtering the Cost Bound at a Variable

At a variable, for a particular candidate value we find the costs of all outgoing constraints. If we have a bound on the cost of the subtree stemming from the variable, clearly the sum of the costs returned from the outgoing constraints must be less than the bound if we are going to improve the solution. Therefore, when one outgoing constraint returns its cost, we subtract the

cost from the current bound. The resultant value is then used as the bound on the cost of the next outgoing constraint. Similarly, if we find a new lowest cost solution with a particular candidate value, that new cost becomes the bound on the solutions using the other candidate values. Figure 18 shows the pseudocode for the filtering of the cost bound at a variable.

At a constraint, similar bound-filtering is done. When a candidate constraint is selected, its RelaxationCost is subtracted from the bound. If the result is less than or equal to 0, any solution using this candidate constraints will not improve on the current lowest cost solution. Therefore, we do not need to value propagate along that candidate constraint. If the result of the subtraction is greater than 0, we propagate it as the new bound. This is because the cost from the subtree below the constraint must be less than the bound minus the cost of the constraint. If the RelaxationCost of all candidate constraints is greater than the bound, a cost of -∞ is returned to indicate a bound failure. The cost propagation mechanism checks for this flag and deals with it appropriately. Figure 19 presents the pseudocode for the filtering at a constraint.

```
current bound = bound passed to constraint
for each candidate constraint{
   result = current bound - local cost
   if result > 0{
      get cost by propagation along candidate constraint with
         the new bound = result
      if cost + local cost < current bound
         current bound = cost + local cost
   }
}
```

Figure 19. Filtering the Cost Bound at a Constraint

4.2.3 Enhancements

A number of additional enhancements are possible to the relaxation schema:

- Bounding the lookahead distance of the value propagation. The value propagation is complete in that it visits all variables connected via outgoing constraints. In larger graphs it may be useful to be able to set a bound on the distance of the value propagation.
- Allowing the behaviour on cycle detection to be specified in the algorithm instantiation.
- Providing a sufficient cost to the search. Whenever a complete solution is found that has a cost that is equal to or less than the sufficient cost, search is stopped. In this way, the designer of the instantiation can specify when a solution is “good enough” to be acceptable.

4.3 Cache-Based Relaxation Algorithms

We propose three incomplete (i.e. they are not guaranteed to find the minimal cost solutions) repair-based relaxation algorithms that use cost caching and search bounding mechanisms. General pseudocode is shown in Figure 20.

For each of the three relaxation algorithms:

- An initial solution is quickly found with a partial, non-backtracking (PNB) algorithm. The relaxation algorithms are therefore repair-based since they try to iteratively improve the initial solution.
- All domain values are used for the candidate values for the source variable.
- The value propagated to an adjacent variable is the only candidate value for that variable. This requires that a relaxation uniquely identify the value for the subsequent variable.[2] Each candidate constraint is satisfied by a single tuple, thus the choice of candidate values is collapsed with the choice of the candidate constraints at the previous constraint.
- All attached constraints are selected as outgoing constraints. The only exception is the constraint that is propagated along to reach the variable.
- The relaxation propagation is done only if the cost of a new solution is lower than the cost of the current solution.

```
find initial solution with PNB algorithm
while(number of iterations ≤ iteration bound){
   choose source variable
   for each value in the domain of the variable{
      temporarily assign the value
      find PostCost by propagating along all constraints
   }
   if (lowest cost < current cost){
      assign value corresponding to lowest cost
      do relaxation propagation
   }
   number of iterations++
}
```

Figure 20. Pseudocode for the Main Loop of the Cache-Based Relaxation Algorithms

We embed instantiations of the relaxation schema within a loop based on the number of iterations. An iteration, therefore, is the selection of a source variable, value/cost propagation for each value in the domain of the source variable, and relaxation propagation if a new, lower cost solution is found. The difference among the three algorithms is how the candidate constraint set is chosen at each constraint. To fully specify each algorithm, we need only define how we find an initial solution and the selection of candidate constraints.

2. In general, the candidate values at a variable need only *satisfy* the relaxations made to previous constraints.

4.3.1 Finding an Initial Solution

An initial solution for the problem is found using the partial, non-backtracking (PNB) algorithm. PNB is a non-backtracking, prospective technique that is the forward step of the PEFC3 algorithm (see Section 2.3, Chapter 2 and below, Section 4.4.2). An *inconsistency count* is kept for each value in the domain of each variable and originally set to 0. The PNB algorithm operates as follows:

1. Randomly choose a variable and assign it to a randomly chosen value.
2. Propagate the value to adjacent variables. For each value in the domain of the adjacent variable, if the value is inconsistent with the assignment, increment the inconsistency count of the value.[3]
3. Select the variable with the largest mean inconsistency count and assign to it the value with the lowest inconsistency count. Break ties randomly.
4. Repeat steps 2 and 3 until all variables are assigned.

The complexity is linear in the number of constraints because there is no backtracking.

4.3.2 Single Min-Value Relaxation

The first relaxation algorithm is the Single Min-Value (SMV) algorithm. SMV selects very few candidate constraints in order to find a solution with smaller time and space complexity. At most two candidate constraints are chosen at each constraint. Propagating across constraint, c_i, from $x_a = v_j$ to x_b using the SMV algorithm, we choose the following candidate constraints:

1. The constraint, c_p, such that **RelaxationCost**(c_p) is the minimum for all constraints that allow $x_a = v_j$. If more than one constraint meets this requirement, one of the minimum cost constraints is chosen randomly.
2. The constraint, c_q, such that **RelaxationCost**(c_q) is the lowest cost of all constraints that allow $x_a = v_j$ and $x_b = v_{other}$, where v_{other} is the value to which x_b is already assigned.

Clearly, it is possible for $c_p = c_q$, and, in this case, only one constraint is selected.

Four orderings for selection of the source variable are used in the experiments: random, most constraints, least constraints, and most violated. The last is found by summing the costs of the constraints attached to the variable. The variable with the highest sum is considered the most violated. With multiple iterations, we further specify that each variable is only chosen once to be the initial variable. The final three selection criteria create an ordering over the list of variables with ties broken arbitrarily.

---

3. The amount that the inconsistency count is incremented is the relaxation cost of the intervening constraint.

Two termination criteria were used: 5 iterations and 10 iterations. Recall that in one iteration, a source variable is chosen, value/cost propagation is done with each of the values in the domain of the source variable, and relaxation propagation is done if a new, lower cost solution is found.

These options produce eight related algorithms.

Variable Selection	Iterations	
	5	**10**
Random	SMVR5	SMVR10
Most Constraints	SMVHL5[a]	SMVHL10
Least Constraints	SMVLH5	SMVLH10
Most Violated	SMVV5	SMVV10

TABLE 1. **Variations of the SMV algorithm**

a. The HL indicates the order is from High-to-Low number of constraints. Similarly, LH indicates Low-to-High.

4.3.3 Multiple Min-Value Relaxation

The Multiple Min-Value (MMV) algorithm is designed to find the cost of a larger number of alternative graph configurations. It is therefore expected to find lower cost solutions than SMV, but to use more effort in finding those solutions. The candidate constraints chosen are:

1. All constraints that allow $x_a = v_j$ and minimize the local cost. As noted above, it is possible for a number of candidate constraints to have the same local cost while allowing $x_a = v_j$.
2. The constraint, c_q, such that **RelaxationCost**(c_q) is the lowest cost of all constraints that allow $x_a = v_j$ and $x_b = v_{other}$, where v_{other} is the value to which x_b is already assigned.

Only one constraint is selected if c_q is the unique minimum cost constraint allowing $x_a = v_j$.

We use the same ordering for source variable selection as used in SMV and two options for termination criteria: 1 iteration and 2 iterations. These termination criteria were used because the complexity of the MMV algorithm (see below) makes more iterations impractical.

Eight algorithms are specified.

Variable Selection	Iterations	
	1	**2**
Random	MMVR1	MMVR2
Most Constraints	MMVHL1	MMVHL2
Least Constraints	MMVLH1	MMVLH2
Most Violated	MMVV1	MMVV2

TABLE 2. **Variations of the MMV algorithm**

4.3.4 MinConflicts-like Relaxation

The MinConflicts repair algorithm chooses a variable with violated constraints, tries instantiating it to all possible values, and, if a lower cost configuration is found, assigns the minimum cost value to the variable. We define a MinConflicts-like (MC) algorithm in our schema. The candidate constraint chosen is:

1. The constraint, c_q, such that **RelaxationCost**(c_q) is the lowest cost of all constraints that allow $x_a = v_j$ and $x_b = v_{other}$, where v_{other} is the value to which x_b is already assigned.

The variable ordering for the source variable is the same as in MinConflicts: most violated.

There are two termination criteria: 20 iterations and 50 iterations. Because the only candidate constraint is the one that makes the current assignments valid, the source variable is the only variable whose value can change in a single iteration. If all variables are used as source variables (in a series of iterations) and no relaxation has been done, then further iterations will not change the solution. The search has found a state where it is required to change the value of two variables in a single iteration to find a lower cost solution. The MC algorithm can not move from this state. We terminate the search when we detect this situation regardless of the number of iterations that have been completed.

These elements specify two algorithms.

	Iterations	
Variable Selection	**20**	**50**
Most Violated	MC20	MC50

TABLE 3. **Variations of the MC algorithm**

The difference between MC and MinConflicts is the propagation. In the usual implementation of MinConflicts, there is no need to follow constraints through the graph since the cost only changes around the variable that is being moved. MC does more work than MinConflicts because it must traverse the entire graph and re-assign to each constraint the candidate constraint to which it is already assigned. This has no effect on the cost of the graphs that MC will produce relative to MinConflicts; however the effort expended in MC will be higher.

4.3.5 Complexity

Both SMV and MMV algorithms have a worst-case time complexity that is exponential in the number of constraints in the problem. The SMV complexity is $O(D2^m)$, where the maximum size of the candidate constraint set is 2, D is the size of the domain for the initial variable, and m is the number of constraints. The complexity of the MMV algorithm is $O(D(max(|CC_k|))^m)$, where CC_k is the set of candidate constraints at constraint, c_k, and $max(|CC_k|)$ is the size of the largest set of candidate constraints chosen for any outgoing constraint.

The complexity of the MC algorithm is linear in D and the number of constraints. Only one candidate constraint is chosen at each constraint; therefore, the each constraint is only traversed once. Cost propagation and relaxation propagation each traverse each constraint once.

4.4 Experiments with PCSPs

We have described the use of information caches to deal with cyclic graphs and decrease the time complexity of the relaxation schema. These techniques show a high space complexity and, in some sense, achieve the improvement in time complexity only by creating space complexity difficulties. We expect that the algorithms defined here will be appropriate only on small problems where the space complexity is manageable. The Partial Constraint Satisfaction Problems (PCSPs) discussed in Chapter 2 are such problems. We now compare performance of the above algorithms to the PEFC3 algorithm on a series of PCSPs.

4.4.1 Partial Constraint Satisfaction Revisited

A partial constraint satisfaction problem is an extension to CSPs [Freuder 92] . The problems are defined so that a CSP solution (where all constraints are satisfied) does not exist due to contradictory constraints. The goal is to find the valuation that fails to satisfy as few constraints as possible. More formally, a PCSP is the following:

Given:

- A set, V, of n variables, $\{x_1, \ldots, x_n\}$, each with a discrete, finite domain $\{D_1,\ldots, D_n\}$.
- A set, C, of m binary constraints $\{c_1, \ldots, c_m\}$ each defined on a pair of the variables in V.

Find:

- An assignment of values to variables (from their respective domains) such that the maximum number of constraints are satisfied.

In modeling PCSPs as a relaxation problem, we define that each constraint, c_i, on variables x_j, x_k, has a compatibility set, CS_i, of the form $\{(x_{j1}, x_{k1}), (x_{j2}, x_{k2}), \ldots, (x_{jp}, x_{kp})\}$. The compatibility set specifies the pairs of values that satisfy the constraint. Constraint checking searches the compatibility set for the value-pair to which the variables are instantiated.

4.4.2 Partial Extended Forward Checking

In forward checking for CSPs, when a variable is instantiated the values of adjacent variables that are inconsistent with the instantiation are pruned. [Freuder 92] adapts forward checking to PCSPs by keeping an *inconsistency count* for each value in the domain of each variable. The inconsistency count represents the cost of assigning the value to that variable. Originally, all inconsistency counts are 0. When a variable is instantiated, any value in the domain of adjacent variables that is inconsistent has its inconsistency count incremented.

The partial extended forward checking algorithm (PEFC3) is the best PCSP algorithm found by [Freuder 92] . PEFC3 estimates a lower bound on the cost of assigning a value by summing the cost of the solution so far with the inconsistency count of the value and the sum of the minimum inconsistency counts for the uninstantiated variables. The last term is the minimum further increase in cost that choosing the value would incur. If this sum equals or exceeds the cost of the best complete solution found so far, the value is pruned from the domain of the variable. Backtracking occurs when a domain is empty. Pseudocode for the forward-step of the PEFC3 algorithm is shown in Figure 21. Backtracking undoes a previous instantiation and correctly restores the domains and inconsistency count of each value.

```
instantiate a variable to a value and update CostSoFar
for all uninstantiated, connected variables{
   for all values in the domain{
      if CostSoFar + InconsistencyCount + Sum of minimum
            InconsistencyCounts from uninstantiated variables ≥
bound{
         prune
         if domain is empty backtrack
      }
   }
   for all remaining values in the domain{
      update InconsistencyCount
      if updated and CostSoFar + InconsistencyCount + Sum of minimum
            InconsistencyCounts from uninstantiated variables ≥
bound{
         prune
         if domain is empty backtrack
      }
```

Figure 21. The Forward Step of the PEFC3 Algorithm

By checking the inconsistency count before updating, values can be pruned earlier. After updating, it is only those values whose inconsistency counts have been incremented that need to be re-checked.

In PEFC3, values are assigned to variables in order of increasing inconsistency counts, while variables are instantiated by decreasing order of mean inconsistency count.

4.5 Experiments

Experiments were performed to compare the proposed relaxation algorithms with the PEFC3 algorithm. We use six sets of PCSP problems with a varying number of variables and varying costs for relaxation. This section describes the evaluation criteria and the problem sets.

4.5.1 Evaluation Criteria

[Freuder 92] evaluates the algorithms by the number of consistency checks necessary to find a solution. In our model, we count the number of times the local cost of a value pair is accessed. This is an exact analogue of the consistency checking. Where a consistency check has to access the constraint definition to find if a particular tuple satisfies the constraint, finding the local cost requires accessing the constraint definition to find the cost incurred at the constraint by a tuple. Because of the format of the algorithms we propose, the local cost of a pair is looked-up whenever one member of the pair is propagated across the constraint. Therefore, we are counting the number of times value propagation is done across any constraint.

The PNB algorithm, used to find an initial solution, and the relaxation propagation both require a single access for each constraint in the graph. These counts are added to that from the relaxation algorithm to produce the results. In the presentation of our results we will, like [Freuder 92] , refer to this measure as the number of consistency checks. All of the relaxation algorithms proposed here are not guaranteed to find the minimal cost solution, therefore we also assess the cost of the solutions that are found.

In all algorithms where a random choice is made (all SMV algorithms, MMVR1, and MMVR2) the results presented are the average from five runs using different random seeds.

4.5.2 Problem Description

Six sets of PCSP problems were used for the experiments. Problem Sets 1, 3, and 5 were randomly generated by [Freuder 92] . Problem Sets 2, 4, and 6 were created from 1, 3, and 5 respectively by giving a random cost in the range [1,9] to each value-pair not in the original satisfiability set.

[Freuder 92] creates many PCSP problems by varying four parameters: number of variables, number of constraints, domain size, and number of value-pairs in the compatibility set. The final three parameters are represented by probability values which are held constant in the experiments used here. The problem parameters are shown in Table 4.

Problem Set	Number of Variables	Expected Density	Expected Domain Size	Satisfiability	Costs
1	10	0.3	0.2	0.4	0-1
2	10	0.3	0.2	0.4	0-9
3	12	0.3	0.2	0.4	0-1
4	12	0.3	0.2	0.4	0-9
5	16	0.3	0.2	0.4	0-1
6	16	0.3	0.2	0.4	0-9

TABLE 4. **Parameters of the Experimental Problems**

In brief, the problems are constructed as follows:

- The number of variables is set and a spanning tree is created to ensure that all problems will contain a single, connected graph.
- The expected density is used to determine the number of constraints added to the spanning tree. A value of 0.3 indicates that 30% of the possible constraints are added to the spanning tree graph.
- Expected domain size is based on a maximum domain size of twice the number of variables. A value of 0.2 means that on average each domain has a size that is 40% of the number of variables ($2 \times 0.2 \times n$).
- Finally, satisfiability indicates the proportion of the tuples in the cross-product of the domains of adjacent variables that appear in the compatibility set of the connecting constraint. A satisfiability value of 0.4 indicates that approximately 40% of the possible value-pairs satisfy the constraint connecting the variables.

Each of last two factors is applied separately to each domain or constraint. None of the problems have a satisfying (i.e. zero cost) solution.

4.6 Results

Because we have 18 variations of relaxation algorithm, rather than comparing each to PEFC3, we first conduct preliminary experiments to compare the variations amongst themselves. The variations with the best results are then compared against PEFC3 on a larger set of problems.

4.6.1 Preliminary Experiments

The preliminary experiments used Problem Sets 1 and 3 to determine the best combination of options for the relaxation algorithms. We focus on the cost results for two reasons. First, the main weakness of the relaxation algorithms, vis-a-vis the PEFC3 algorithm, is incompleteness. If the algorithm does not find a near-optimal solution, then it is not reliable enough for use regardless of the reduction of effort. Secondly, the incomplete algorithms do significantly less work than the PCSP algorithm in almost all cases.

In both the SMV and MMV algorithms, there was little systematic difference among the variations. We were prevented from assessing the full effect of the manipulations in the termination criteria because, in a large number of problems, the algorithm with the lower iteration bound found an optimal solution. It was observed that the SMVLH algorithms often found a solution of equal or lesser cost than any others and, in problems where improvements could be realized by further iterations, the largest cost decreases were seen with SMVLH. For these reasons SMVLH5 and SMVLH10 are compared with PEFC3 below.

Even less variation was found among the versions of the MMV algorithm. There is a cost difference among algorithms on only 4 of the 20 problems. In those cases, it was the MMVLH algorithms that exhibited solutions of lesser cost than the other algorithms. Therefore, we will include MMVLH1 and MMVLH2 in our comparisons.

No difference was observed in the cost of the solutions found by the two MC algorithms. In 75% of the problems, however, an optimal cost solution was not found. Comparing the constraint checking data shows that in all but 3 problems the number of constraint checks is identical between the two versions. This indicates that the secondary termination criterion (trying all values of all variables without finding a lower cost solution) was active in most problems. In other words, if any improvement on the cost of the solution can be found, it is found within 20 iterations and moreover, the MC algorithm often finds a local minima where two or more values must be simultaneously changed in order to find a lower cost solution. MC only changes at most a single value in an iteration so it cannot escape from this situation. We will not compare the MC algorithms further due to their cost performance.

4.6.2 Consistency Checks

Figure 22 shows the average number of consistency checks for algorithm on each problem set. Note the log scale on the vertical axis. The MMV algorithms were not able to find solutions to the problems in Sets 5 and 6 due to their exponential growth in memory requirements.

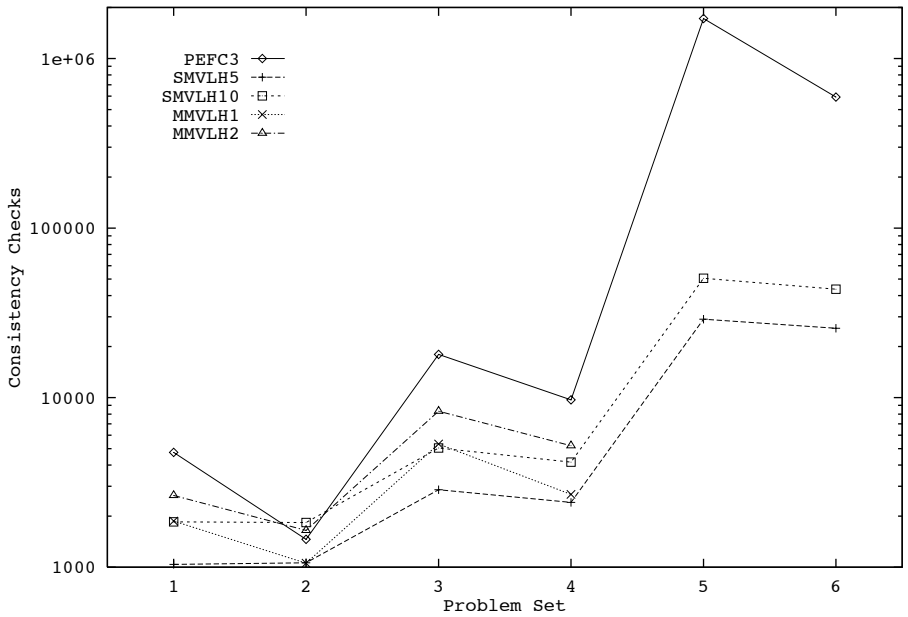

Figure 22. Comparison of the Number of Consistency Checks for SMV, MMV, and PEFC3 Algorithms

The relaxation algorithms use significantly fewer consistency checks than the PEFC3 algorithm in all but the second problem set. The difference in number of consistency checks is more pronounced with the problems having a higher number of variables. In almost all cases solutions to the problems with relaxation cost greater than or equal to 1 (Problem Sets 2, 4, and 6) are found more quickly than the solutions to the corresponding unit-cost problems (Problem Sets 1, 3, and 5, respectively).

4.6.3 Solution Cost

Figure 23 presents the percent difference from optimal for all the relaxation algorithms. The PEFC3 algorithm is complete and so guaranteed to find the optimal solution. Note again that the MMV algorithms could not find solutions to the problems in the final two sets due to memory requirements.

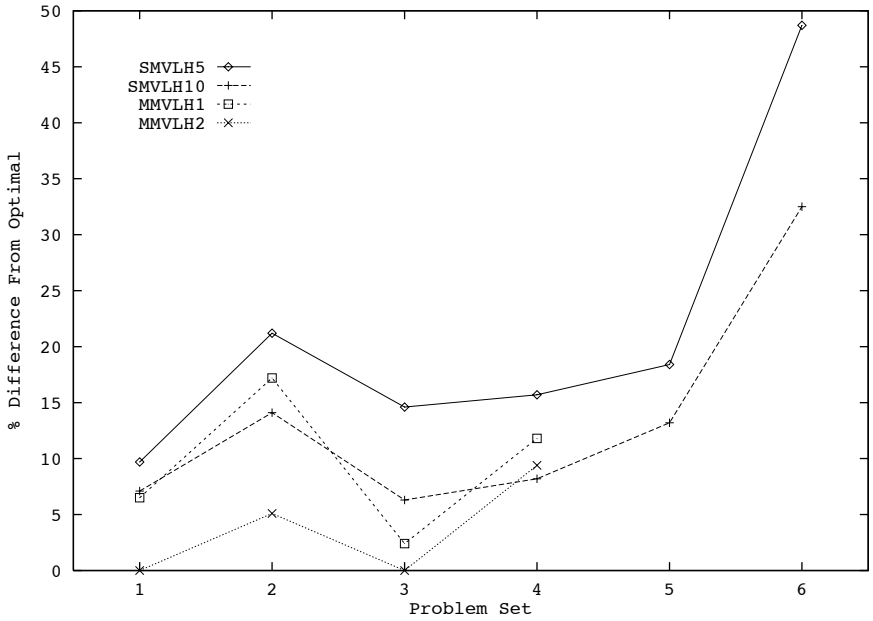

Figure 23. Percentage Difference from Optimal Costs for SMV and MMV Algorithms

The cost of the solutions for each algorithm on the odd-numbered problem sets are within 20% of the optimal for all relaxation algorithms. In fact, on problem sets 1 and 3, the MMVLH2 algorithm found the optimal solution for all problems. Despite the significant reduction in effort for the relaxation algorithms, the solutions found are close to the optimal in most situations. For the even-numbered problem sets we see a wider variance in the solution cost results, especially in sets 2 and 6. In the latter set, the SMVLH5 algorithm is only within 50% of the optimal. Finally, note that as the number of variables increases, the cost of the solution found by the relaxation algorithms is further from the optimal.

4.7 Discussion

These experiments demonstrate the usefulness of the relaxation schema in PCSP problems. The results indicate that the algorithms that are instantiations of the schema expend less effort than PEFC3, the best PCSP algorithm found by [Freuder 92] , in almost all problems. At the same time, the cost of the solutions found is close to or even equal to the optimal in a number of the problem sets. We feel strongly that the time-to-solution is a key element in the utility of the solution, and that in some cases a non-optimal, quickly found solution is more desirable than a perfect solution found too late. This is obvious when the size of problems increases to the point where optimal solutions can not be found at all in a reasonable amount of time. The generality of our approach and the flexibility of the schema allows specification of algorithms of varying forms of complexity, allowing the tailoring based on the time pressure.

There are a number of more specific observations and conclusions to be drawn from these experiments. Each of the sections below focuses on a particular aspect of the results.

4.7.1 Non-Uniform Relaxation Costs

The effect of a variety of relaxation costs can be seen in the respective comparison of each even-numbered problem set to the preceding odd-numbered set. The consistency check results demonstrate that almost all algorithms do less work on the problem sets with multiple relaxation costs than on the unit-cost problems. The solution cost results indicate that the relaxation algorithms perform worse on the multiple cost problems than on the unit cost problems.

4.7.1.1 Consistency Checks

The reduction in effort in the PEFC3 algorithm is due to the removal of a number of values from variable domains before they are assigned. With unit-cost relaxations, the inconsistency cost of a value inconsistent with a single current variable assignment, is incremented by 1. There will typically be a number of values with the same inconsistency count. The order of instantiation among values with the same inconsistency count is arbitrary, therefore the algorithm is likely to run through a number of them before finding the optimal value. In contrast, with multiple relaxation costs, there is a wider range of inconsistency counts because the counts are incremented by a value between 1 and 9. It is more likely that the values with lower inconsistency costs will be part of the optimal solution, therefore the inconsistency count is a better value ordering. Fewer values will be tried and discarded before the optimal value for a variable is found.

The reduced effort in the MMV algorithms on the problems with multiple relaxation costs is due to the smaller size of the candidate constraint set. The candidate constraint set is composed of all minimum cost relaxations consistent with previous instantiations. With unit-valued relaxations, the candidate constraint set will be composed of all zero-cost relaxations or, if there are no consistent zero-cost relaxations, all relaxations with a cost of 1. With multiple

relaxation costs, the relaxation costs range from 1 to 9. The set of consistent minimum cost relaxations will therefore be smaller. As with the inconsistency costs of PEFC3, the multiple relaxation costs allows a better ordering over the choices. In the MMV algorithms, it allows the a smaller candidate constraint set to be chosen at each variable.

The reduction in the number of consistency checks is less pronounced for the SMV algorithms. In fact, more consistency checks are done for Problem Set 2 than in Problem Set 1. The reductions from Set 3 to Set 4 and Set 5 to Set 6 are nominal. The argument used for the MMV algorithms does not apply since the size of the candidate constraint set is 2 or less for the SMV algorithm. The range of relaxation costs does effect when the search can be cut-off early, however. With the possibility for a large difference in cost between the two selected candidate constraints, it is more likely that the cost of a solution found with the less expensive constraint will be exceeded earlier in the subsequent search using the more expensive constraint. Therefore, less search is done. This effect is also present in the MMV algorithms, but as demonstrated by the SMV data, the reduction in effort is small.

4.7.1.2 Solution Costs

The poorer performance of the relaxation algorithms on the multiple cost problems is related to the fewer consistency checks observed for the same problems. It is possible to have an optimal solution with a small number of expensive relaxations rather than a large number of low cost relaxations. The globally optimal relaxation may not be the local minimum cost relaxation. Given the wider range of relaxation costs, this is more likely in the problems with multiple relaxation costs. The reduction in effort seen above goes hand-in-hand with a poorer cost performance: by searching through fewer possible relaxations (due to the wider cost range), the incomplete relaxation algorithm has a greater chance of missing the optimal relaxation at a variable. In the unit cost problems, more possibilities are explored because of the difficulty of differentiating among relaxations on the basis of local costs.

It is to be expected that when the heuristic for choosing relaxations to explore picks a smaller number of relaxations, the average quality of the solution decreases because it is more likely to miss critical relaxation decisions.

4.7.2 Increasing the Number of Variables

As the number of variables in the problems increase, we observe an increase in the work done by all algorithms and a decrease in the cost-performance of the relaxation algorithms. The trend in effort is not surprising given the exponential time complexity of the algorithms. More interesting is the poorer cost performance of the relaxation algorithms.

In the description of the composition of the problems (Table 4, Section 4.5.2), it is noted that the number of constraints is relative to the number of variables in the problem. The larger number of constraints will, on average, lead to a higher cost optimal solution because more constraints have to be relaxed. Indeed, this is reflected in the average optimal solution costs for the problems sets, shown in Table 5. The multiple cost problem sets show a similar trend.

Problem Set	Average Number of Constraints	Average Optimal Cost
1	19.2	3.1
3	26.6	4.1
5	48.2	7.6

TABLE 5. **Average Number of Constraints and Average Optimal Cost for each Problem Set**

With a larger optimal cost, the local relaxation costs are less informative. The global impact of a relaxation of a particular constraint is still the key piece of knowledge. However, the ability of the local cost of the relaxation to predict the global impact is lessened. In the low cost problems (e.g. an optimal cost of 3), the local cost (e.g. a relaxation cost of 1) is a large part of the global cost. Therefore, it is likely that the local minimum cost will be the global minium cost relaxation. Intuitively, the lower the local cost, the lower the global cost will be. In contrast, when the optimal cost is higher (e.g. 8), the local cost is a relatively smaller part of the global cost and it is less likely that the minimum local relaxation is the optimal relaxation due to interactions with the rest of the graph. The minimum local cost heuristic is less successful. This explanation predicts that a on problems with a large number of constraints and a low cost solution, the relaxation algorithms should find close to optimal solutions.

An alternate explanation for the lessened cost performance is that with a larger number of constraints, there are more decisions to be made. Given that the decision-making is not complete and that many of the decisions may be critical to finding the optimal solution, there are more opportunities for an incorrect decision to be made. The more incorrect decisions, the further from optimal will be the solution.

We believe that both these explanations are at work and in fact are complementary. Not only is the number of imperfect decisions to be made increased as indicated by the latter argument, but also the imperfectness of the decisions is increased as predicted by the former explanation. Investigation of the effect of larger graphs on cost performance and on these particular explanations is an area for future work.

4.7.3 Estimation of Global Cost vs. Local Cost

The PEFC3 algorithm uses propagation to gather more global information on the impact of assigning a particular value. The global information is used to make an estimation of the lower-bound on the cost increase resulting from a particular value instantiation. The relax-

ation algorithms are also aggregating more global information in the form of the costs that are propagated following value propagation. The difference between the algorithms, at this level, is that the PEFC3 algorithm is using its global information as a variable and value ordering heuristic and backtracks in attempt to find an even lower cost solution than the current. The relaxation algorithms use the local relaxation cost information in order to select the candidate constraints that will be searched over.

This observation suggests that if there is a relatively inexpensive source of global information, it will be a benefit to the relaxation algorithms to base the selection of candidate constraints on that information. Texture measurements can be just such a source of information. Given, the large decrease in processing effort in the relaxation algorithms, another area of future work is the improvement of the algorithms by the integration of texture measurements to supply an estimate of more global information. Such an integration with the relaxation algorithms detailed here would involve the selection of candidate constraints based on dynamically calculated texture measurements.[4]

4.7.4 The MC Algorithm

The variations on the MC algorithm performed poorly on the preliminary experiments so they were not compared with the PEFC3 algorithm on the full set of problems. Given that the MC algorithm is closely based on MinConflicts, this result is somewhat surprising.

It is known that repair-based algorithms need a “good” starting solution in order to perform well [Davis 94] . If the original solution is a local minima requiring two variables to change value to find a lower cost solution, then the repair will not succeed in moving from the starting solution. The PNB algorithm, because it assigns the lowest cost value for a variable with no backtracking, may find such a minima. We conducted experiments on all six problem sets using a random starting solution. Cost performance was better, but still between 40% and 70% above optimal for 50 iterations.[5] The MC algorithms did fewer consistency checks than each of the other algorithms.

Given that the poor performance is not simply an artifact of the PNB starting solution, we analysed the original data on each problem. Unfortunately, it was not the case that MC performed well on some problems and very poorly on others. In only a few problems in each set could the MC performance be identified as good or poor.[6] No consistent pattern relating performance difference to problem structure differences was in evidence.

---

4. In Chapter 5, we will use texture measurements to limit selection of outgoing constraints in much larger graphs (e.g 100 variables).

5. Experiments were also done with 100 and 200 iterations. There was little difference in cost performance.

6. We defined MC to have exhibited “good” performance when it was within 25% of the optimal cost and “poor” performance when it was above 75% of the optimal cost. About four problems per set could be categorized into one of these two performance categories.

All the problems share the structure defined by the parameters of expected density, expected domain size, and satisfiability (see Table 4, Section 4.5.2). In the original PCSP experiments [Freuder 92] , these parameters were varied and each was found to have a statistically significant interaction with the number of consistency checks done in the complete algorithms investigated. Because the algorithms were complete, no statistical analysis could be done on the interaction between cost performance and these parameters. We postulate that such interactions do exist and further research, with a wider set of PCSPs, may give indications as to underlying structural explanations for the poor performance of the MC algorithm.

4.8 Summary

This chapter has presented cache-based techniques for algorithms that instantiate the relaxation schema. Three algorithms were created and two of them fared well in empirical comparison to the best algorithm for PCSPs. Due to the exponential time and space complexity of these algorithms, we do not view it likely that they will be applicable to any problems approaching the complexity of real-world problems. However, our approach allows specification of relaxation algorithms tailored to the time pressures and constraint types of the problem domain. In the following chapter, we investigate such limited algorithms in the domain of schedule optimization.

Chapter 5 Relaxation-Based Estimation of the Impact of Scheduling Decisions

In this chapter, we demonstrate how relaxation-based methods can be used for estimating the impact of scheduling an activity at a particular start time. We discuss job shop scheduling and the constraint-based representation for such problems. This is necessary background for the elucidation of the estimation algorithms. Three texture-based estimation algorithms are proposed and compared against a baseline estimator on a set of medium-sized scheduling problems (i.e. 100 activities, 5 resources).

5.1 Estimating the Impact of a Scheduling Decision Revisited

In Chapter 2 (Section 2.5.4), we discussed techniques to estimate the global impact of assigning an activity to a start time. The key point in that work is that when attempting to schedule an activity, it is desirable to know the impact that assigning a start time will have on the overall schedule cost. Dynamically calculated estimates have been shown to be superior to static or dispatch rule based estimates in a number of situations [Sadeh 91] . Estimates will be calculated many times during scheduling, so we require the calculation to be efficient.

GERRY [Zweben 92] uses the number of local conflicts as an estimate for the global conflicts created by an assignment. In contrast, MICRO-BOSS [Sadeh 91] propagates the changes in the marginal cost to unscheduled activities. The method for identification of the best reservations for an order is applicable only to the cost model used in the experimental problems. Furthermore, the propagation of information occurs only within an order. Potential increase in costs caused by interaction with other orders via resource constraints are not taken into account. Earlier work on the propagation of preferences through a temporal constraint graph has shown that the complexity of full propagation is prohibitive [Sadeh 89] . The preference propagation only addressed temporal constraint graphs and so, like the marginal cost propagation in MICRO-BOSS, it to could not aggregate preference information over resource constraints. The only example of which we are aware of using both inter- and intra-order information is the MRP-STAR system [Morton 86] . The iterative calculation used is expensive and does not scale-up well to multiple resources and activities.

We believe that the ability to efficiently, accurately estimate the cost of scheduling decisions is a key to finding good schedules. Further, we hypothesize that the ability to dynamically select the subgraph over which cost gathering will be done, independent of constraint type and on the basis of texture measurements, will improve the accuracy and efficiency of estimates.

We model the sources of cost in a schedule by the cost of relaxations necessary to satisfy constraints. We, then, estimate the impact of the assignment of a value by the value and cost propagation phase of our relaxation schema. The generality of the schema allows us to aggregate information from all parts of the graph (that is, via any type of constraint) and the ability to specify the actions at the decision points allows us to integrate heuristic decision making to minimize the overhead associated with full propagation.

5.2 Job Shop Scheduling

The job shop scheduling problem is a common, NP-complete [Garey 79] problem in the literature [Fox 87] [Sadeh 91] [Nuitjen 93] [Davis 94] . The basic format is as follows:

Given:

- A set of orders where each order has a release date and a due date.
- A process plan for each order containing one or more activities with precedence relationships among the activities in the order and defining resources required by each activity.
- A set of unit capacity resources.

Assign a start time to each activity such that:

- All precedence relationships are maintained.
- No resource is used by more than one activity at any time point.[1]

5.2.1 A Constraint Model for Job Shop Scheduling

A number of authors [Fox 87] [Sadeh 91] [Ijcai 93] have approached job shop scheduling by using constraints to represent both the precedence relationships and the resource requirements of activities. Each activity is represented by combinations of variables and constraints.

Our constraint model will build on the model used in the ODO scheduler [Davis 94] . In this section we present an overview of the activity-level representation. The following sections will define the variable and constraints types introduced here. A diagram of two activities that require the same resource is shown in Figure 24.

1. Most scheduling models define a scheduling horizon, a time granularity, and a number of time points within the horizon.

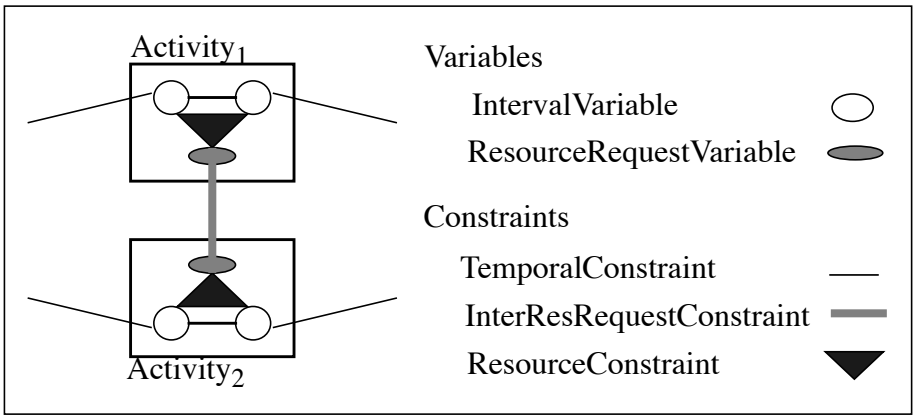

Figure 24. Schematic Diagram of The Activity Representation

Each activity is composed of three variables: StartTime and EndTime which are IntervalVariables and Resource which is a ResourceRequestVariable. The StartTime and EndTime variables are connected by a TemporalConstraint. All three variables within an activity are connected by a ResourceConstraint. Figure 24 also shows constraints between activities. The TemporalConstraints are used to represent the precedence constraints by connecting IntervalVariables in different activities of the same order. The InterResRequestConstraint connects ResourceRequestVariables, in different activities, that use the same resource. These variable and constraint types can be divided into those representing the temporal dimension and those representing the resource dimension.

5.2.2 Temporal Variables and Constraints

An IntervalVariable has a value that is a time point. The domain is represented as a closed interval of time points, indicating that the variable can take on any value in that interval including the end points.

A TemporalConstraint is used to express a precedence constraint between two variables. The basic precedence constraint in job shop scheduling specifies that activity, A_j, occurs after another activity, A_i, has completed. For activities A_i and A_j (Figure 25), we specify that $\text{EndTime}_i + \alpha = \text{StartTime}_j$, where $\alpha \geq 0$. We represent α by the *value* of the constraint. The value is an interval of time points where each time point in the interval is a possible value for α.[2]

We link the EndTime_i with the StartTime_j via a TemporalConstraint, TC_1. The value of TC_1 is $[0, \infty]$ as shown. To satisfy TC_1, the difference between the value of the StartTime of A_j and the value of EndTime of A_i must be an element of the interval value.

2. Equivalently, any element of the interval value is an acceptable difference between the values of the constrained variables.

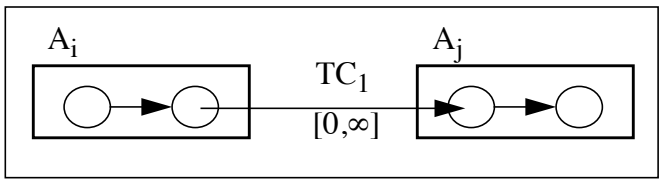

Figure 25. A Simple Precedence Constraint Between 2 Activities

The value of a constraint explicitly defines the temporal relationship enforced by the constraint. The interval value allows representation of a wide range of temporal relationships by appropriately setting the upper and lower bounds of the interval. TemporalConstraints can be combined to represent each of Allen's time relations [Allen 83] . The DURING and OVERLAPS relationship are shown in Figure 26. The TemporalConstraints point in the direction of the higher-valued variable so, for example, the StartTime of $Activity_2$ must equal the StartTime of $Activity_1$ plus an element in the interval $[1, \infty]$.

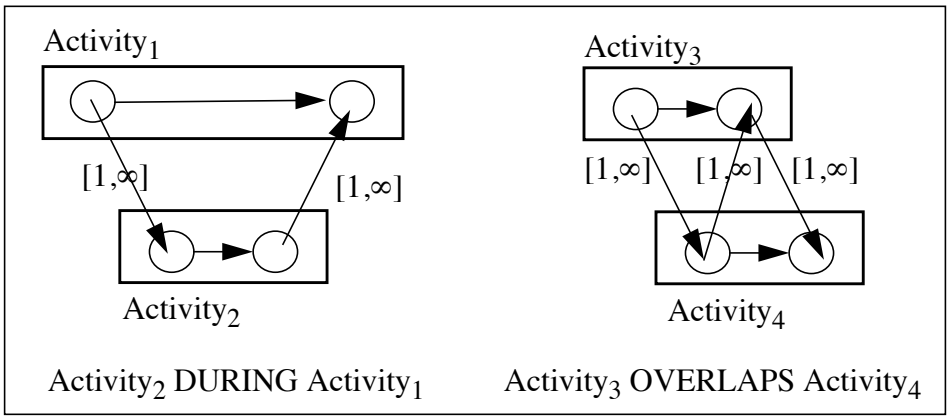

Figure 26. A Representation of Two Allen Relations with TemporalConstraints

5.2.3 Resource Variables and Constraints

Variables and constraints are also used to model the resources required by activities in job shop scheduling. Each resource has a finite capacity. Two activities that use the same unit-capacity resource, can not overlap in their execution. We use ResourceRequestVariables, ResourceConstraints, and InterResRequestConstraints to model the capacity requirements.

A ResourceRequestVariable must represent both the resource required by the activity and the time over which the resource is required. The temporal value is an interval of time points. The resource value indicates the name of one of the resources defined in the problem.

A ResourceConstraint connects two IntervalVariables and a ResourceRequestVariable. It is an equality constraint that is satisfied only when the temporal value of the ResourceRequestVariable is equal to the interval between the values of the IntervalVariables. For example, the temporal value associated with variable v_3, in Figure 27, must be equal to $[v_1, v_2]$.

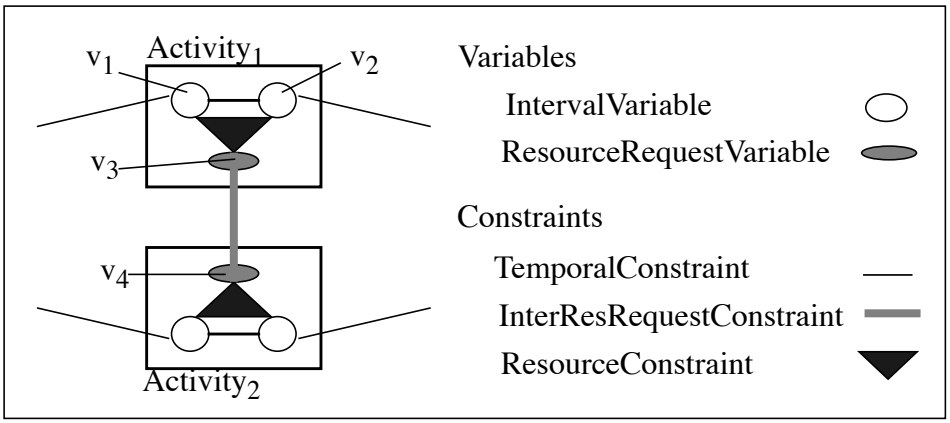

Figure 27. Schematic Diagram of The Activity Representation

An InterResRequestConstraint joins only ResourceRequestVariables that have the same resource value. The constraint is satisfied when the temporal intervals at the connected ResourceRequestVariables do not overlap. Because an InterResRequestConstraint exists between v_3 and v_4 in Figure 27, the variables must have the same resource value.[3] For the constraint to be satisfied the temporal values at v_3 and v_4 must not overlap.

5.2.4 Example

Figure 28 displays a diagram of the complete representation for a very small job shop scheduling problem.

The problem specifics are as follows:

- Scheduling horizon: [0, 100].
- 3 orders: O_1, O_2 and O_3.
- Each order, O_i, has two activities: A_{i1}, A_{i2}, each with a duration of 10. The exception is A_{31} which has a duration of 20.
- 2 resources: R_1, R_2.
- A_{12}, A_{22}, and A_{31} use R_1, A_{11}, A_{21}, and A_{32} use R_2.
- Earliest release date: 0.
- Latest acceptable due date: 100.

3. The representation of alternative resources is achieved by the relaxation of resource InterResRequestConstraints. A relaxation that allows the use of another resource would actually change the connectivity of the graph. If the alternatives are equally preferred, the relaxation cost can be specified as 0.

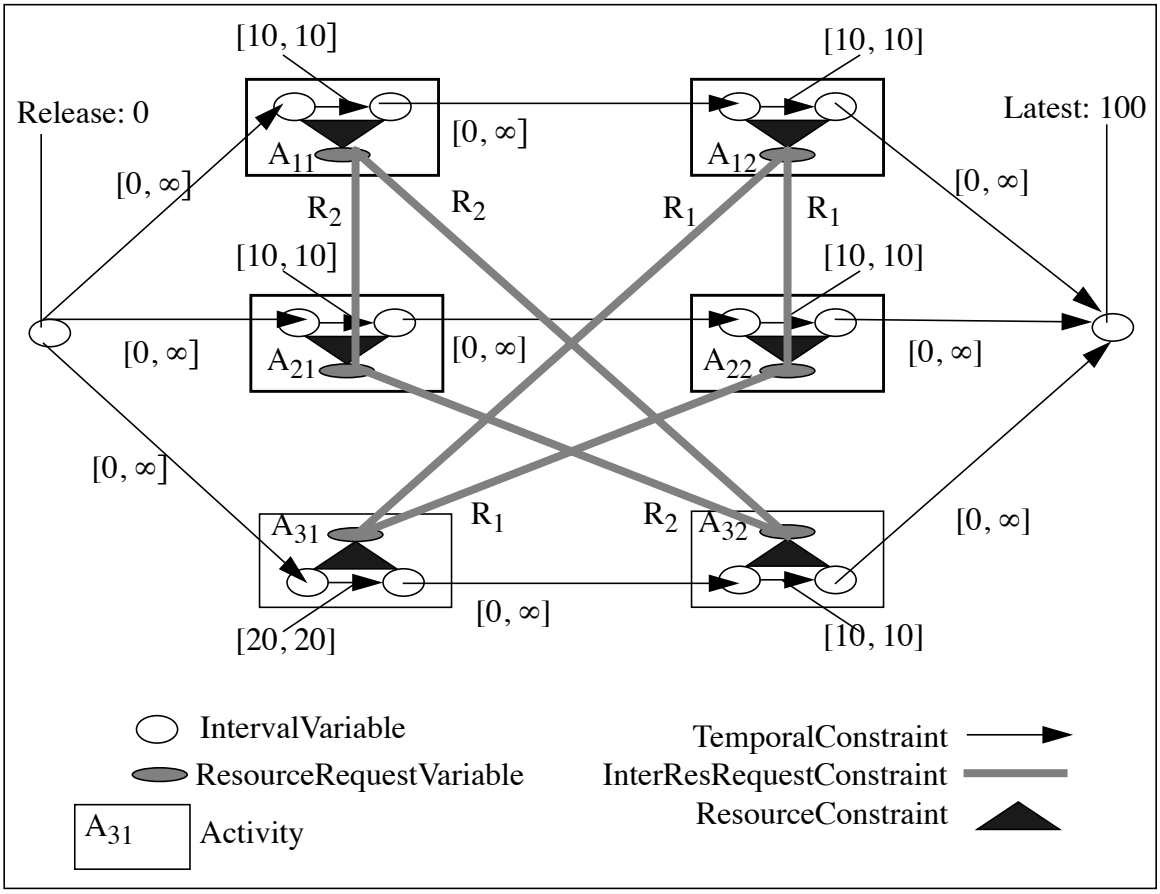

Figure 28. Constraint Representation of a Simple Job Shop Scheduling Problem

5.2.5 Constraint Relaxation in Scheduling

As discussed in Chapter 1 (Section 1.2.3) and Chapter 2 (Section 2.5), scheduling is an optimization problem on criteria such as tardiness, inventory, and throughput. We explicitly model the sources of cost in a scheduling problem with relaxable constraints. Previous work in ISIS and MICRO-BOSS schedulers has taken this approach. For example, due date constraints could be relaxed up to the latest acceptable due date for the order. Our general constraint relaxation work can be applied to not only model optimization criteria like tardiness and inventory costs but also sophisticated resource-based trade-offs. For example, it may be that a particular activity has different possible durations. The long duration corresponds to using a machine at a slow speed with minimal machine wear, whereas the small duration corresponds to using the machine at high speed with increased wear. The long duration activity costs less (due to maintenance costs). This can be represented as a relaxable duration constraint where the RelaxationCost function is inversely proportional to the duration. We believe

that most (or all) currently used optimization criteria can be modeled in this fashion and that it has the generality to represent many more criteria. Further research is required to assess the validity of this belief.

We only address temporal constraint relaxation in this dissertation. For the problems studied here, all other constraint types are non-relaxable. We leave the definition of complex cost models via relaxation of other constraint types for future work. TemporalConstraints are based on the value of the constraint represented by an interval as described above (Section 5.2.2). Relaxation is simply a widening of this interval. The GenerateRelaxation function, therefore, returns a set of constraints that have a *wider* interval value than the original constraint. For each TemporalConstraint, the RelaxationCost function is defined to be a function of the widening of the constraint.

5.2.6 Extending the Example

Figure 29 shows the problem from Figure 28 with the addition of relaxable constraints modeling tardiness and inventory costs. The non-relaxable constraints have been removed from the diagram for clarity. We model tardiness with a TemporalConstraint between the EndTime of the last activity in an order and an IntervalVariable representing the due date. The value of the constraint is $[0, \infty]$. If the EndTime is less than or equal to the due date, no relaxation need be done. Otherwise, the constraint interval is expanded, incurring some cost.

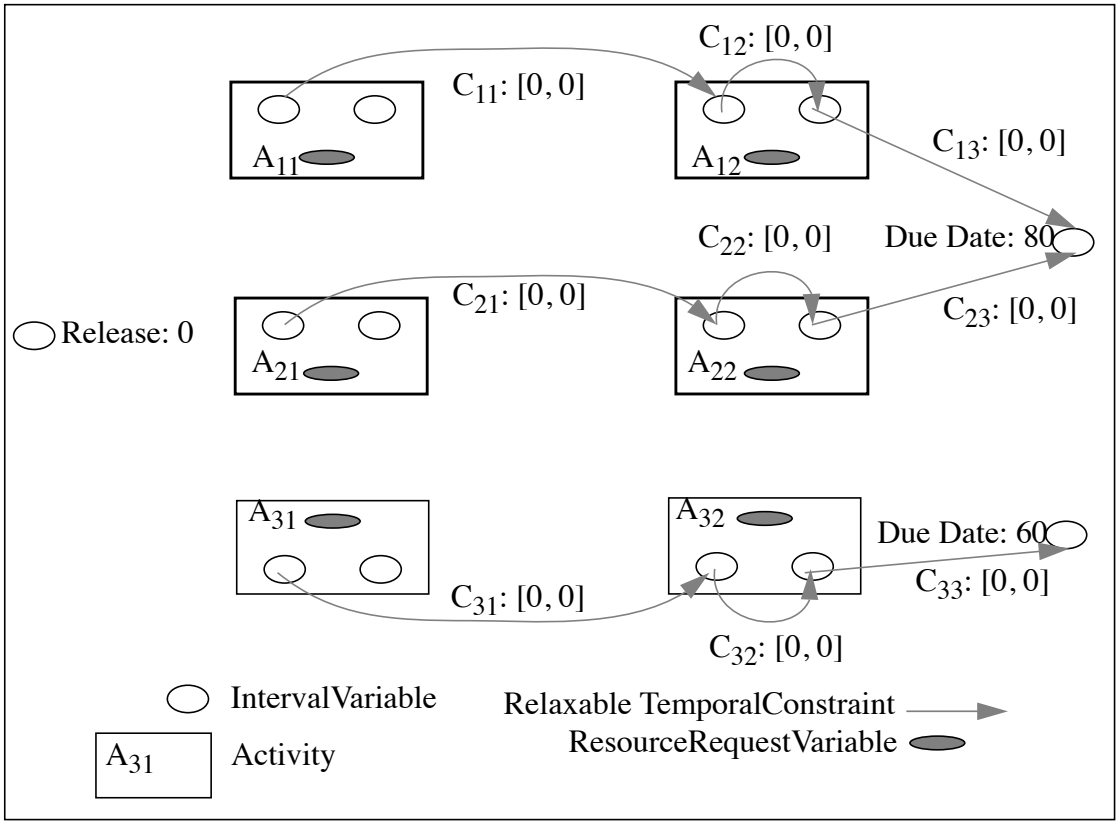

Figure 29. Optimization Constraints on the Example Scheduling Problem

The due date on O_1 and O_2 is 80 and on O_3 is 60. Note that the due date (in Figure 29) is different from the latest acceptable due date (in Figure 28). This pattern of optimization constraints is a slight simplification of the one used in the experiments below. Note that the due date constraints have the value [0,0] rather than [0,∞]. This is due to the fact that we are optimizing the inventory cost as well as tardiness and so wish to model costs if the order completes before or after the due date.[4]

As an example of relaxation, let us focus on constraint C_{13}. For the constraint to be satisfied, the end time of A_{12} must be 80. If we need to relax the constraint to allow A_{12} to complete after the due date by 10 time points, we modify the value from [0, 0] to [-10, 0].

5.3 A General Relaxation-Based Algorithm for Cost Estimation

To find the minimum cost impact of assigning an activity to a start time, we could generate all schedules in which that activity was assigned to that start time. The minimum global cost of the schedule reflects the minimum cost impact of that particular assignment. The complexity of such a method clearly makes it impractical. We will use texture measurements on the constraint graph to identify a salient subgraph for an activity. *We hypothesize that the impact of a scheduling decision on the subgraph is representative of the impact on the entire graph.*

5.3.1 The General Estimation Algorithm

We propose estimation algorithms based on our relaxation schema. Given a *source activity* and a value for the source activity, the texture measurements identify a subset of activities within the graph. A number of solutions are produced for this subgraph by Monte Carlo simulation and the lowest cost solution is used as an estimate for the impact of the value.

Two parameters are used in the algorithms to govern the number of solutions found:

- NumberOfSets - the number of sets of solutions for a single value of the source activity.
- SetSize - the number of solutions that make up one set of solutions.

For each set of solutions, the first solution is found by a constructive search algorithm. The subsequent solutions in the set are found by randomly changing the value of one of the scheduled activities. After each set of solutions, the order of instantiation of the activities is randomly mixed to allow the next iteration of the constructive search to find a different solution.

Given a source activity and a value for that activity, the algorithm is as follows (see Figure 30 for a diagram):

---

4. In the cost model for our experiments, we use two constraints between the EndTime of the final activity and the due date. The two constraints represent tardiness and inventory costs, respectively.

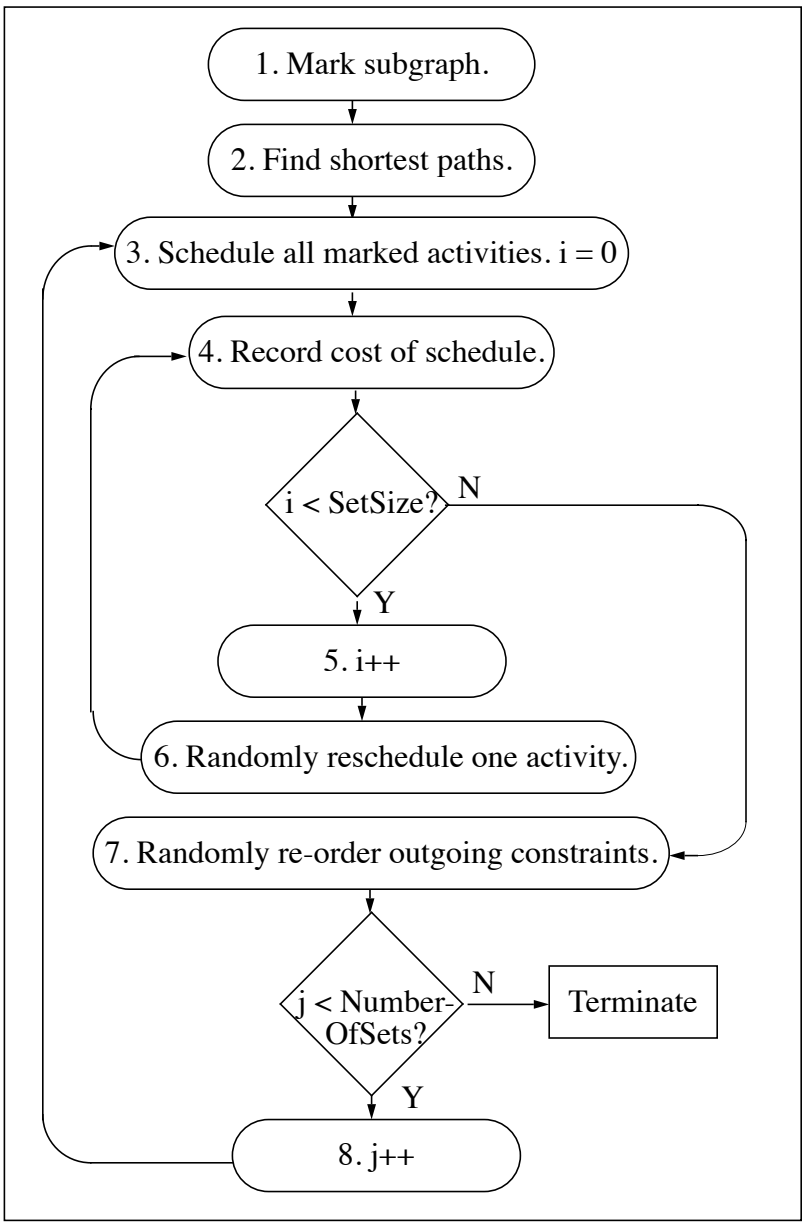

Figure 30. Flow Chart of the General Estimation Algorithm

1. Mark a subset of the activities in the scheduling problem based on some criteria. Typically, we base the marking decision on a texture measurement.
2. Find the shortest paths from the source activity to each marked activity. Label the constraints on each path as outgoing constraints from the activity closer to the source activity.
3. Schedule each marked activity with the constructive algorithm. Set the counter, i, to 0.
4. Find the cost of the solution. If the new cost is less than the current lowest cost, store the new cost.
5. Test if $i <$ SetSize. If so, increment i and continue to 6. If not, jump to 7.

6. Randomly change the value of a scheduled activity, loop to 4.
7. Randomly re-order the outgoing constraints at the source activity.
8. Test if counter j < NumberOfSets. If so, loop to 3. If not, terminate.

A number of points arise from the specification of this general algorithm:

- The algorithm operates on activities rather than variables. The constraints within an activity are not relaxable. We can treat the graph as a network of activities and constraints rather than as a network of variables and constraints. This abstraction realizes a three-fold reduction in the number of nodes in the graph.
- In step 3, all the activities in the subgraph are assigned a start time. A constructive algorithm is used to find a consistent solution for the subgraph. See Section 5.3.3 below.
- The outgoing constraints at the source activity are randomly re-ordered before each use of the constructive algorithm. The solutions generated by randomly changing activity assignments are relatively close together in the search space. By re-ordering the outgoing constraints, we change the order of instantiation of the activities and so move to a remote position in the search space. We gather further random solutions in the new area.
- The total number of solutions found for each candidate value of the source activity is SetSize × NumberOfSets.
- The cost at an activity is a texture measurement in itself. It is a measurement of the constraint tightness surrounding the activity. The higher the cost, the larger the necessary relaxation and therefore, the tighter the constraints were before relaxation. By aggregating these costs for each value of the source activity, we are able to find an estimate of the global impact of the assignment.

It remains to specify the following techniques before turning to the specific algorithms:

- Identifying the subgraph.
- Finding the solution with constructive search.
- Calculating the cost of the subgraph schedule.

5.3.2 Finding a Subgraph

The preprocessing on the graph consists of the marking a subset of the activities in the graph and identifying the shortest path between the source activity and each marked activity. Each constraint on this path is defined to be an outgoing constraint of the activity that is closer to the source activity. Each path of outgoing constraints, beginning at the source activity, terminates at a marked activity. The criteria for marking activities varies among the algorithms.

Though all the marked activities appear in the subgraph, there may also be some unmarked activities in the subgraph (i.e. activities on the shortest path between the source activity and a marked activity). Only the marked activities are scheduled by the constructive algorithm.

More formally, the entire scheduling problem is represented as a graph $G = (V, E)$, where V is a set of vertices (activities) and E is a set of edges (constraints). The preprocessing finds a subgraph $G' = (V', E')$, $V' \subseteq V, E' \subseteq E$. All the elements in V that are marked are also in V', however there may be some elements in V' that are not marked.

5.3.3 The Constructive Search Algorithm

To find the first solution, we begin with the source activity assigned to a candidate value and a subgraph connected by the outgoing constraints. The constructive algorithm is as follows:

1. Value propagate along all non-relaxable constraints (in the whole graph, G) to prune inconsistent values in the domain of other activities.
2. Propagate control from the source activity, along one of the outgoing constraints to the adjacent activity.
3. Assign the activity to the earliest consistent start time and value propagate along all non-relaxable constraints in G. If no such value exists return to previous activity (go to 5) with the backtrack flag set.
4. If there are outgoing constraints that control propagation has not visited, go to 2. Otherwise, return control to the previous activity (go to 5) with the backtrack flag un-set.
5. At the previous activity, if the backtrack flag is set, prune the current value from the domain, go to 3. Otherwise, go to 4.

5.3.3.1 Value and Control Propagation

In all the previously presented relaxation algorithms, a variable is instantiated and the control and value propagation proceed together to a connected variable along an outgoing constraint. As noted in Chapter 3 (Section 3.2.1), value and control propagation are more generally independent.

When an activity is assigned, value propagation proceeds along all temporal and resource constraints (all constraints, c, such that $c \in E$). Elements in the domain of adjacent variables are pruned if they are not consistent with the assignment. If a value is pruned, the domains of other activities may also need to be pruned. The value propagation may proceed to, at most, all activities in the graph (all activities, a, such that $a \in V$). The value propagation ensures that when a marked activity is assigned a value, that value is consistent with previous assignments.

Control is propagated along each outgoing constraints in some sequence and the activities reached are instantiated to a consistent value. We arbitrarily choose the lowest consistent value in the domain. The only exception to this is the constructive search with the Contention/Reliance algorithm (see Section 5.5.6.2). If no consistent value exists, chronological backtracking is done.

5.3.3.2 Backtracking

Chronological backtracking is initiated when an activity has no consistent value to which it can be assigned. That activity returns control to the previous activity with a flag indicating the failure. The previous activity unassigns itself and undoes the effect of the value propagation.[5] The activity then chooses another consistent value and performs value and control propagation as usual. If the activity does not have any more consistent values, it returns control to the next-most recently assigned activity, indicating the need to backtrack.

Backtracking is, at worst, exponential in time complexity and so is prohibitively expensive.The expense is compounded by that fact that the backtracking algorithm may be used a number of times to find an solution for each value of the source activity. It is critical that very little or no backtracking occur.

5.3.3.3 The Relaxation Schema

In terms of our relaxation schema, we have the following specification for each heuristic decision point:

- Source activity: Chosen *a priori*.
- Candidate values: All consistent values. The Monte Carlo simulation randomly samples the candidate values in finding the subsequent solutions.
- Outgoing constraints: All constraints identified in the preprocessing step.
- Candidate constraints: All outgoing constraints are non-relaxable, therefore only candidate constraint at each outgoing constraint is simply the constraint itself.
- Value-commitment: We are gathering costs; no value-commitment is done.

5.3.4 Calculating the Cost of a Subgraph Solution

Relaxable constraints are the source of cost in our model of schedule optimization, therefore the cost calculation is based on an estimate of the necessary relaxations. Cost propagation is performed after a solution has been found for the subgraph, at which point control is at the source activity. The cost propagation passes control along the outgoing constraints and at each marked activity the cost of all attached, relaxable constraints is calculated. The sum is then propagated back toward the source activity where it is summed with all the other costs.

Each relaxable constraint is marked when its cost is calculated. This prevents the double counting of relaxation costs for constraints connecting two marked activities. After all marked activities have calculated their local cost, the cost of any relaxable constraints that have not been marked is calculated. This cost is added to the sum of the local costs. The final sum reflects an estimate of the cost of the whole graph given the partial schedule.

---

5. In undoing the effects of the value propagation, the activity must restore the graph to the state it was in before the activity was assigned the previous value. We use a context stack to cache search states to which we may backtrack. There is no search involved in undoing the value propagation of a previous assignment.

Because we are scheduling a subset of the activities in the full problem, we have the following three cases in the calculation of the relaxation cost of a single relaxable constraint.

1. Both connected activities have been scheduled.
2. Only one of the activities has been scheduled.
3. Neither activity has been scheduled.

In the first case, the constraint simply relaxes itself to allow the two assigned values to be consistent. In the second case, the domain of the unassigned activity is examined and, of the remaining consistent start times, the start time that will *maximize* the relaxation cost is chosen as a temporary value. Relaxation is then performed with this temporary value. Finally, if neither activity is assigned, the domains of each activity is analyzed and the start times that will maximize the relaxation cost of the constraint are used. For example, if the constraint has an interval value of $[0,\infty]$, the maximum cost occurs when the activity at the lower end is assigned to its maximum value and the other activity is assigned to its minimum value.

In the latter two cases we examine the domains of the unscheduled activities to find a worst case estimate of the relaxation cost incurred at the constraint. We choose the maximum cost in order to find an upper bound on the impact of the value assigned to the source activity. It is possible however, that the domain of one of the connected activities is empty; that is, there are no consistent candidate values. This occurs when the activity is not one of the marked activities and is not assigned. In that case, we use the worst possible case relaxation cost based on the original domains of the activities.

We choose the maximum cost relaxation when we have no other information about the relaxation to be made for two reasons.

1. We want the estimates to be an upper bound on cost of the relaxations. A true upper bound would allow the estimates to be used in other algorithms where a choice must be made between activities and not simply about values within an activity. For example, we may want to make the activity/value selection a single decision point and so must judge between a number of activity/value tuples to make an assignment. Unfortunately, even using the worst case costs does not guarantee a true upper bound because the subgraph solution may not be a partial solution to the larger graph (see Section 5.3.5).
2. A subgraph solution is not guaranteed to be a partial schedule for the whole problem. By using pessimistic estimates, we hope to counteract the likelihood that some of the cost estimate will be based on subgraph solutions that will not lead to an overall solution.

5.3.5 Partial Solutions vs. Solutions to the Subgraph

The constructive search algorithm finds solutions to the subgraph. The solutions are not necessarily partial solutions to the whole graph. It is possible that one or more of the unmarked activities will have empty domains in a solution found by the constructive search. Backtracking only occurs when an attempt is made to schedule a *marked activity* and no valid start times

exist. Because the unmarked activities are not scheduled, an empty domain will not be recognized. This is the desirable behaviour because we are attempting to find a fast estimate of the impact of a schedule decision rather than actually scheduling.

5.3.6 Complexity

If no backtracking is done, the general complexity of the algorithm depends on the value propagation. During the constructive algorithm, whenever an activity is assigned, value propagation will, at most, visit all activities in the complete graph. The number of activities instantiated during the constructive algorithm is, at most, the size of the subgraph. If there are a total of n activities in the graph, and m activities in the subgraph, the value propagation has a time complexity of $O(mn)$. The instantiation of the activities in the constructive search (with no backtracking) incurs an additional complexity of $O(m)$ and the number of random assignments made after each backtracking solution is SetSize - 1. The whole loop is performed NumberOfSets times, therefore the overall complexity to find cost estimate for one value of an activity is $O(\text{NumberOfSets} \times (mn + m + \text{SetSize} - 1))$. If backtracking is necessary the worst case complexity of constructive search is $O(d^m)$, where d is the maximum domain size.

5.4 The Cost Estimation Algorithms

There are four methods by which the subgraph is selected. The first method, the Depth algorithm, is used for a baseline estimate of the actual impact. We compare the other algorithms to the Depth algorithm. The other three algorithms depend on texture measurements to identify the activities with the largest effect on the cost of a candidate value at the source activity.

5.4.1 The Depth Algorithm

The Depth algorithm is used to establish a baseline cost estimate against which the other algorithms are compared. The algorithm is designed to choose a subset of activities based on simple-minded criteria. The Depth algorithm marks all activities that are directly connected to the source activity via a path of constraints of length L or less.

5.4.2 The TemporalOnly Algorithm

The TemporalOnly algorithm marks those activities that are connected to the source activity via a path of temporal constraints. This has the effect of strictly restricting the size of the subgraph over which solutions are found. In our experimental problems, there are a total of 5 activities in an order and only the activities within an order are connected via temporal constraints. The number of marked activities for the TemporalOnly algorithm is 4.

This algorithm is designed to estimate the impact of activity reservations in a way similar to the marginal cost propagation in MICRO-BOSS. The algorithm should not be interpreted as a re-implementation of marginal cost propagation as it is both less efficient and more general.

5.4.3 The TopCost Algorithm

The TopCost algorithm uses the *expense* texture measurement based on the maximum cost that might be incurred by an activity. As described below (Section 5.5.2), the RelaxationCost of the relaxable constraints is a function of the width of the relaxation and the weight (or CostFactor) of the constraint. Because the RelaxationCost varies directly with the CostFactor, the higher the CostFactor of a relaxable constraint, the more potential effect it has on the overall schedule cost. We identify the expense of an activity as the sum of the CostFactors of its relaxable constraints. Intuitively, the more expensive activities have a large impact on the overall cost schedule and therefore provide an accurate estimation of the impact of an assignment.

For the TopCost algorithm, we use the expense texture measurement to identify and mark the *m* most expensive activities (other than the source activity).[6]

5.4.4 The Contention/Reliance Algorithm

The Contention/Reliance algorithm uses a different texture measurement to identify activities with a large impact on the cost of a scheduling decision. The time intervals over which there is the most contention for a resource and the activity that is most reliant on the resource for that time interval are identified. These texture measurements were developed for and used as a basis of the activity ordering heuristic in MICRO-BOSS. In overview, we identify the *m* most contended for resource intervals and mark the activities most reliant on each interval.

The contention of an activity for a particular resource at a start time is defined in equation 3.

$$Contention = \frac{1}{|d'|} \quad \textbf{(3)}$$

Where:

d' is the domain of consistent candidate values at the activity.

$|d'|$ is the size of the domain of consistent candidate values at the activity.

Using equation 3, we build a profile of the contention of an activity for a resource across all consistent values of that activity. An activity with very few consistent values will have a high contention (i.e. close to 1) for each of its consistent start times. We sum the profiles of each activity on a resource to form an aggregate resource contention profile. An aggregate resource contention profile is generated for each resource.

Based on an interval size equal to the average duration of an activity on a resource, we identify the top *m* intervals in terms of high aggregate contention for any resource. For each interval, we identify the most reliant activity. The reliance of an activity is defined to be the area under its individual contention curve over the interval. The most reliant activity of an interval

6. The source activity may be one of the *m* most expensive activities. In this case, the next most expensive activity is marked and the source activity is left unmarked.

is therefore the activity that contributed most to the aggregate resource contention. We mark the most reliant activity at each interval. If an activity is the most reliant activity on more than one interval, we mark the activity and discard all but one of the intervals where it is most reliant. We then find the next most contended for intervals in order to bring the interval total back to m. This ensures that a total of m unique activities are marked.

The assignment of the source activity and the subsequent value propagation significantly prunes the domains of some activities. Resource contention depends on the possible values of each unassigned activity, therefore the marking of the activities is performed for every value of the source activity. This differs from the Depth, TemporalOnly, and TopCost algorithms, where the marked activities are independent of the value of the source activity.

5.4.5 Expectations

The performance of each algorithm will be evaluated in comparison to the baseline estimate. Assuming the baseline estimate is accurate (see Sections 5.7), we expect the following results.

- Controlling for subgraph size and parameter settings, we expect the TopCost algorithm and the Contention/Reliance algorithm will outperform the TemporalOnly algorithm. The TemporalOnly texture measurement is not actually based on information indicating that the activities have a large impact on a scheduling decision. We expect that the algorithms that use the extra information will perform better than the TemporalOnly algorithm. The performance comparison between the TopCost algorithm and Contention/Reliance algorithm is not obvious prior to experimentation.
- We expect that, other settings being equal, the algorithms using larger graphs will outperform the same algorithms on smaller graphs. With a larger number of activities with which to make an estimate, it seems likely that the estimate will be more accurate.
- On graphs of the same size and within the same algorithm, we expect that estimates will be superior when more solutions are generated with the constructive algorithm. The constructive solutions for a single source activity value use different orderings of marked activities and therefore should sample a wide range of locations in the search space. Conversely, the randomly generated solutions will cluster around the constructive solution from which they are produced. Therefore, with a low number of constructive solutions and a lot of random solutions, the quality of the sampling of the search space will be less than with a high number of constructive solutions and fewer random solutions.

5.5 Experiments

We now turn to specific experiments that were run. We first describe the experimental problems, the cost model used, and the evaluation criteria for the algorithms. This is followed with a discussion of how the source activity and its candidate values were selected. Finally, we present an analysis of the accuracy of the baseline estimates and a discussion of the parameter settings used for each algorithm.

5.5.1 Experimental Problems

The experiments use problems in eight problem sets created by [Sadeh 91] . Each set contains 10 scheduling problems and each problem contains 20 orders, 5 activities per order, and 5 resources. Each activity in an order uses a different resource. The main variables in the generation of the problems were the average tightness of due dates, the range of due dates, and the number of bottleneck resources.

- The average tightness is used to set the due date of each order. The higher the tightness, the less likely it is that an order can finish on time. Problems in Sets 1, 2, 5, and 6 have a loose average tightness, while the rest have a tight average tightness.
- The range of due dates specifies the variation in due dates across orders. With a wide range, each order may have a due date far from the due dates of other orders. The narrower the range the more likely that due dates will be close together, increasing the likelihood of conflict among orders. Problems in Sets 1, 3, 5, and 7 have a wide due date range, while the other problems have a narrow due date range.
- A bottleneck resource is one for which there is a high contention over some time interval. A problem with more bottleneck resources will typically be more difficult to schedule. In half of the problems sets (Sets 5 through 8) there are two resources for which there is a high contention over the entire scheduling horizon, where as in the rest of the problem sets there is only one such resource.

5.5.2 Cost Model

The optimization criteria considers both tardy costs and inventory or holding costs. The total schedule cost is the sum of the tardy and holding costs for each order. It is the same cost model used in MICRO-BOSS [Sadeh 91] .

The tardy cost for an order is the product of the TardyCostFactor and the difference between the completion date and the due date. If an order is completed early, no tardy cost is incurred. Each order has an individually specified TardyCostFactor. The tardy cost for order i, $TARD_i$, is as follows:

$$TARD_i = TardyCostFactor_i \times max(0, C_i - dd_i) \quad \textbf{(4)}$$

Where:

C_i is the time of completion of the last activity in order i

dd_i is the due date for order i

The holding costs represent the cost of storing raw materials and intermediate products. A HoldingCostFactor is defined for each activity. The holding cost for a activity is the product of its HoldingCostFactor and the difference between the completion time of the order and the

start time of the activity. If the order finishes early, the holding cost is the product of the HoldingCostFactor and the difference between the due date and the start time of the activity. The holding cost for an order i, $HOLD_i$, is the sum of the holding costs for its constituent activities.

$$HOLD_i = \sum_{j=1}^{n_i} HoldingCostFactor_{ij} \times [max(C_i, dd_i) - ST_{ij}] \quad \textbf{(5)}$$

Where:

n_i is the number of activities in order i

ST_{ij} is the start time of activity j in order i

The holding cost for an order can be equivalently re-written as:

$$HOLD_i = \sum_{j=1}^{n_i} [HoldingCostFactor_{ij} \times (C_i - ST_{ij})] + \left(\sum_{j=1}^{n_i} HoldingCostFactor_{ij}\right) \times max(0, dd_i - C_i) \quad \textbf{(6)}$$

To represent this cost model via relaxable constraints, we define a **CostFactor** (or weight) at each constraint and specify that the RelaxationCost is the product of the CostFactor and the amount that the interval is widened.

We add three types of temporal constraints to the problem representation:

1. A TardyConstraint connects the final activity of each order to a fixed variable with the value of the due date of the order. The CostFactor of the constraint is the TardyCostFactor for the order. The interval value of the constraint is $[0, \infty]$. The relaxation cost of a TardyConstraint for an order is equivalent to equation 4.
2. A HoldingConstraint connects the start time of each activity in an order to the end time of the final activity in the order. The CostFactor for the constraint is the HoldingCostFactor for the activity. The interval value for a HoldingConstraint is $[-\infty, 0]$ reflecting the fact that cost will be incurred only if the start time of the task is less than the completion time of the order.[7] The sum of the relaxation costs of the HoldingConstraints for an order is equivalent to the first addend in equation 6.
3. An EndHoldingConstraint connects the final activity in the order to the variable representing the due date. The CostFactor is the sum of the HoldingCostFactors for each activity in the order and, as with HoldingConstraints, the interval value is $[-\infty, 0]$. The relaxation cost of the EndHoldingConstraint is equivalent to the final addend in equation 6.

7. With non-zero activity durations, it is clearly impossible to have the start times of activities in an order greater than or equal to the completion time for that order. Therefore, all holding cost constraints will have to be relaxed to some extent. A zero cost schedule is impossible.

5.5.3 Evaluation Criteria

Estimations of the cost of the candidate values of an activity are used to establish the value ordering heuristic in typical scheduling algorithms. It is not necessary that a cost estimation algorithm find estimations that are close to the actual cost. Rather we require that the same ordering be created given the estimated costs as would be created if the actual costs were available. Because the cost estimates are only compared to each other in forming the ordering, a good estimator finds costs such that the difference between the actual costs and the estimated costs is constant across all candidate values. Such an estimator will produce a cost curve that is a vertical translation of the actual cost curve. We define the estimation gap, *EstGap*, of an algorithm on a single problem as follows:

$$EstGap = \sigma(CostDifferenceVector) \quad \textbf{(7)}$$

Where:

σ is the standard deviation of a vector.

CostDifferenceVector is a vector defined by equation 8.

$$CostDifferenceVector = BaselineVector - EstVector \quad \textbf{(8)}$$

Where:

BaselineVector is the costs calculated by the Depth algorithm across all candidate values.

EstVector is the costs calculated by the estimation algorithm across all candidate values.

The estimation gap is a measure of the variability in the difference in estimates between the baseline and the algorithm being evaluated. A value of 0, corresponding to a constant difference between the baseline and the estimations, is optimal.

The complexity analysis above indicates that the number of assignment and the number of activities visited during cost propagation are the relevant factors. Therefore, to compare algorithms on time performance we use the *ChangeActivity* statistic. It counts the number times that an activity is assigned to some value plus the number of activities that are visited in the value propagation during the backtracking algorithm. On this basis the forming of a solution by the random changing of the value of an activity will increment the *ChangeActivity* statistic by one. In contrast, the generation of a solution with the backtracking algorithm will increment the *ChangeActivity* count by, at most, the number of marked activities times the total number of activities in the schedule.[8] The worst case occurs when each instantiation of a marked activity requires that all activities in the graph are visited by value propagation.

8. Again, we assume that no backtracking is done.

Note that each of the texture measurements incur some processing time. We will not include the processing time needed to identify the subgraphs in the *ChangeActivity* statistic as we want to focus on the work done in finding the estimation rather than in selecting the subgraph. This is typically a valid abstraction as the differences in effort in forming the subgraph between texture-based algorithms are small.

5.5.4 The Source Activity

5.5.4.1 Selecting the Source Activity

To this point we have not specified how the source activity is selected. The estimation algorithms are designed to estimate the cost of the scheduling options of any activity, however, an artifact of the experimental problems prevents us from using random selection.

The scheduling problems were created so that only the first and last activities in an order have non-zero CostFactors on their relaxable constraints. Though the underlying cost model allow a holding cost to be specified for each activity, in the experimental problems the holding costs for the entire order are introduced by the first activity. The holding cost constraints for all other activities have a CostFactor = 0. The tardy costs only depend on the end time of the final activity. Therefore, the middle three activities in an order do not have an impact of the cost of the schedule, except indirectly, via interactions with other activities. Furthermore, all the activities using a bottleneck resource have the same relative positions in their respective orders. For example, the third activity in each order may use the bottleneck resource. This pattern of resource use is not used for non-bottleneck resource activities. With this problem structure, it is possible in some problems to find an activity that has no non-zero cost relaxable constraints and, more seriously, that has few or no neighboring activities with non-zero cost relaxable constraints. In such a situation, the cost estimate of one of our algorithms (the Depth algorithm, Section 5.5.5) would be based solely on the worst case costs of all relaxable constraints. The algorithm would find a constant-value cost curve. This is clearly not desired as our results depend on the accuracy of the baseline estimates found by the Depth algorithm.

To avoid this situation, we use the *expense* texture measure (see Section 5.4.3) to identify the most expensive activity. We use the most expensive activity as the source activity in all problems. This activity has non-zero cost relaxable constraints and so will either be the first or last activity in an order. If the activity uses a bottleneck resource, all other activities using the resource will have non-zero cost relaxable constraints (they too will be first or last activities). If the source activity does not use a bottleneck resource, between 25% and 66% of all connected activities will have non-zero cost relaxable constraints. The choice of the most expensive activity is motivated by the desire to ensure the accuracy of the baseline estimates.

5.5.4.2 Selecting Candidate Values for the Source Activity

In a scheduling situation where none of the other activities have assignments, all values in the domain of the source activity are potential assignments. For the purpose of our experiments, we gather a cost estimate for every tenth value of the source activity, starting with the lowest start time. The size of this candidate value set is approximately 30. Sampling the domain of the source activity allows us to build a coarse-grained cost curve over the candidate values. More fine-grained cost estimates can then be made around the lower cost values. For the present experiments, we form only the coarser cost curve.

5.5.5 The Depth Algorithm

The depth algorithm requires a specification of the value of L, the length of paths radiating from the source activity. We experimented with $L = 2$ corresponding to a subgraph of approximately 80 activities (80% of the entire graph). With a subgraph of this size, significant backtracking is noted in the constructive search. No solutions could be found in a reasonable time.[9] Therefore, we use a value of $L = 1$. The subgraph is composed of 20 or 21 activities (20% of the whole graph), plus the source activity in our experimental problems.

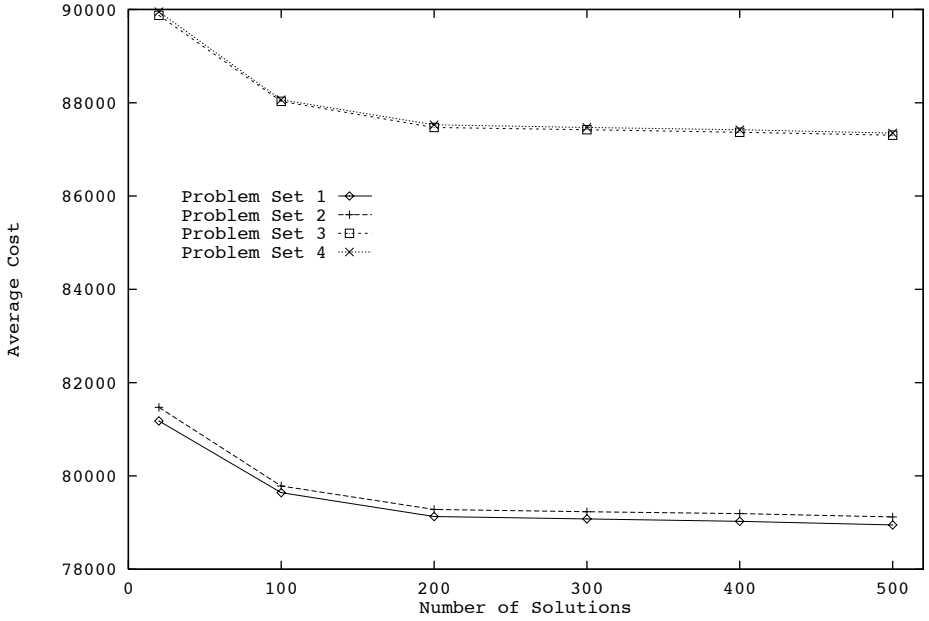

Figure 31. The Average Cost Found by the Depth Algorithm with $L = 1$ for the Problem Sets with 1 Bottleneck

9. The running time on a DECStation 5000/244 on a single problem is estimated to be over 640 hours.

In order to assess the accuracy of the estimate of the Depth algorithm, we ran a series of experiments on all problems. We set the SetSize parameter to 20 and varied the NumberofSets from 1 to 25. Each set of solutions is composed of a single solution found by the constructive algorithm and 19 randomly produced solutions based on the constructive solution. A total of NumberOfSets × SetSize solutions are produced. The x-axis in our plots shows the total number of solutions calculated for each problem in the set rather than the value of NumberOfSets.[10] Figures 31 and 32 present the average cost for each problem set.

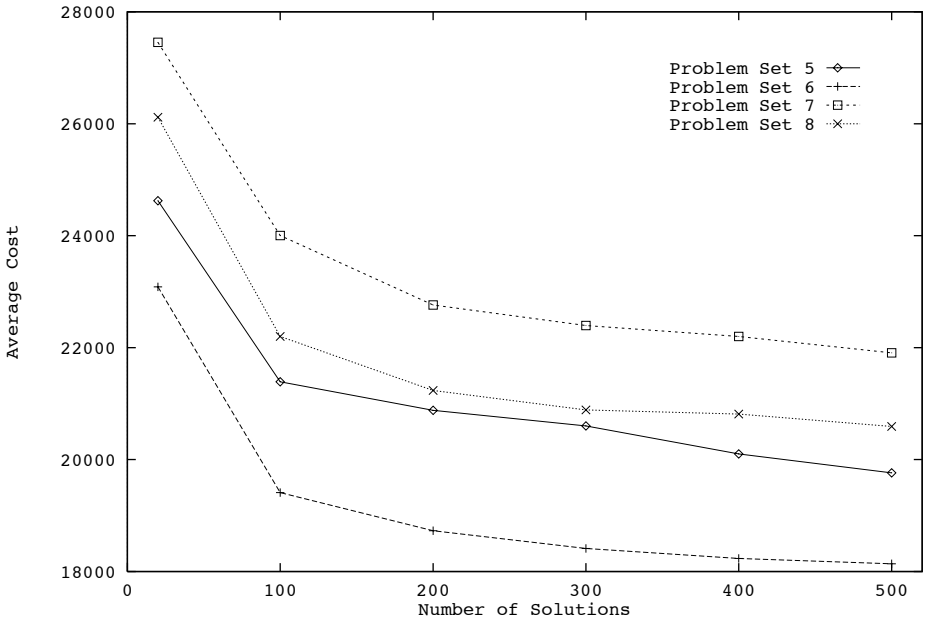

Figure 32. The Average Cost Found by the Depth Algorithm with $L = 1$ for the Problem Sets with 2 Bottlenecks

Two observations are made from Figures 31 and 32:

- The rate of reduction in the average cost lessens with the increase in the number of solutions. With 500 solutions (25 constructive solutions and 475 random solutions), the curves in Figure 31 appear linear and in Figure 32 are close to linear. The small decrease in average cost indicates that further iterations would decrease the cost only slightly.
- The average costs for the problems with 1 bottleneck are much larger than those for the 2 bottleneck problems. Given the structure of the problems and the subgraph, if the source activity requires a bottleneck resource, 19 of the activities in the subgraph also require the same resource. Furthermore, due to the method by which the source activity is selected (see Section 5.5.4.1), each of those 19 activities will have a non-zero cost impact on scheduling decisions. If the source activity is not a bottleneck activity, fewer activities in the subgraph will have non-zero cost impacts. Given that there are the same number of activities with

10. Because SetSize is 20, it is easy to show that the total number of solutions is simply 20 × NumberOfSets.

non-zero impact, when the source activity is a bottleneck activity, a greater percentage of the activities with non-zero impact will be in the subgraph. With 2 bottlenecks the source activity has a greater chance of being a bottleneck activity, therefore the worst case estimate will be used for fewer constraints, and the cost estimates will be lower.

Given these results, we adopt the parameter values of 20 and 25 for SetSize and NumberOfSets, respectively. The cost curves indicate that little significant change will occur to the average costs with more iterations, however we do not have a measure of the accuracy of the baseline estimates as found by the Depth algorithm. In our results section we will present the estimates of the texture-based algorithms in relation to the baseline estimates. We will return the issue of the accuracy of the baseline estimate in the discussion below (Section 5.7).

5.5.6 Parameter Settings for the Texture-Based Algorithms

The parameters that we vary in our experiments are the size of the subgraph over which the estimations are taken and the number and type of solutions that are found.

5.5.6.1 Subgraph Size

Of the three texture-based algorithms, the subgraph size can only be varied in the TopCost and Contention/Reliance algorithms. The TemporalOnly algorithm selects all activities connected via a path of *temporal* constraints. In the experimental problems, only activities within an order share temporal constraints, therefore, with only 5 activities per order, the size of the subgraph is limited to 4 (not including the source activity).

For the TopCost and Contention/Reliance algorithm, we set the size of the subgraph to two different values: 4 and 10. The first size allows comparison with the TemporalOnly algorithm at a constant size. The subgraph size of 10 is approximately half of the size of the subgraph used by the Depth algorithm with $L = 1$. We expect that estimations on this subgraph will be accurate but to do significantly less processing than the Depth algorithm.

5.5.6.2 NumberOfSets and SetSize

We use two combinations of NumberOfSets and SetSize parameters. The first sets NumberOfSets and SetSize to 5 and 20, respectively, while the second sets the parameters to 1 and 500. This manipulation assesses the hypothesis that a number of constructive solutions will perform a better sampling of the search space and provide more accurate estimates.

We were not able to find estimates with the Contention/Reliance algorithm when the parameters were set to 5 and 20. Recall that the Contention/Reliance algorithm finds a subgraph of the activities most reliant on the most contended for activities. This texture measure was originally used as a variable ordering heuristic in MICRO-BOSS, where it was found that instantiating the more reliant activities was a useful heuristic. Our difficulty stems from the fact that when we attempt to instantiate the most reliant activities in an order other than decreasing reli-

ance, significant backtracking is observed. By ignoring the reliance measure as a activity ordering heuristic, we see performance on the constructive algorithm that is much worse than if activities had been chosen at random. Furthermore, the activities selected by the Contention/Reliance algorithm often constrain each other, and therefore the scheduling of one of the marked activities removes all values from the domain of another activity. This behaviour is aggravated by arbitrarily choosing the lowest consistent value in the constructive search.

To avoid the subgraph scheduling problem and significant effort in backtracking, we order the activity instantiation for the Contention/Reliance algorithm in descending reliance and randomly choose a consistent value from the domain in the constructive search. The Contention/Reliance algorithm is run with the NumberOfSets and SetSize parameters set to 1 and 500.

This difficulty indicates that the identified activities represent a difficult portion of the scheduling problem (this was the original intent in the creation of these texture measurements [Sadeh 91]). In terms of the quality of the cost estimates from this subgraph, the difficulty in scheduling the subgraph indicates the importance of these activities to finding a feasible solution which in turn suggests that they will have an important impact on the scheduling decisions. Therefore, we take this difficulty as independent evidence that the Contention/Reliance algorithm will provide accurate cost estimates.

5.5.6.3 Summary of Algorithm Variations

Our algorithms are based on three texture measurements, two graph sizes, and two settings of the parameters governing the number of solutions that are produced. Table 6 shows a summary of the algorithm variations.

Name	Texture Measurement	Graph Size	Parameter Settings
TopCost10(100,20)	Most Expensive	10	100, 20
TopCost10(500)	Most Expensive	10	500, 500
TopCost4(100,20)	Most Expensive	4	100, 20
TopCost4(500)	Most Expensive	4	500, 500
C/R10(500)	Most Reliant	10	500, 500
C/R4(500)	Most Reliant	4	500, 500
TemporalOnly(100,20)	Temporally Connected	4	100, 20
TemporalOnly(500)	Temporally Connected	4	500, 500

TABLE 6. **Variations of the Cost Estimation Algorithms**

5.6 Results

We now present the results of the cost estimation experiments. The first section below presents raw data on three randomly selected problems. With the intuition as to the results provided by the raw data, we move to the comparison of the effects of texture measurements, graph size, and parameter settings. All the comments in these sections are based on the assumption that the baseline estimate is an accurate portrayal of the actual cost impact. We will examine the validity of this assumption in the discussion section below (Section 5.7).

5.6.1 Raw Data

The *EstGap* performance measure is the result of statistical manipulation on the experimental data and so does not provide an intuitive feel for the raw data. Therefore, we randomly choose three problems and display the actual data (for a number of the algorithms) in Figures 33 through 35.The TemporalOnly and TopCost data presented are from experiments with NumberOfSets and SetSize parameters set to 5 and 20 respectively. The data for the C/R algorithms are from experiments with NumberOfSets and SetSize set to 1 and 500, respectively.

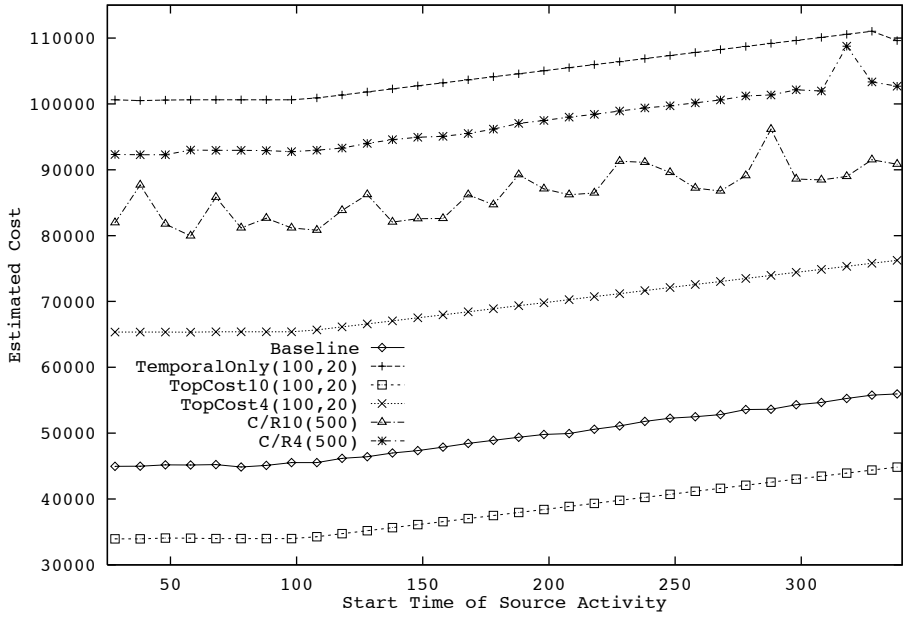

Figure 33. Raw Data for Problem 3, Problem Set 2

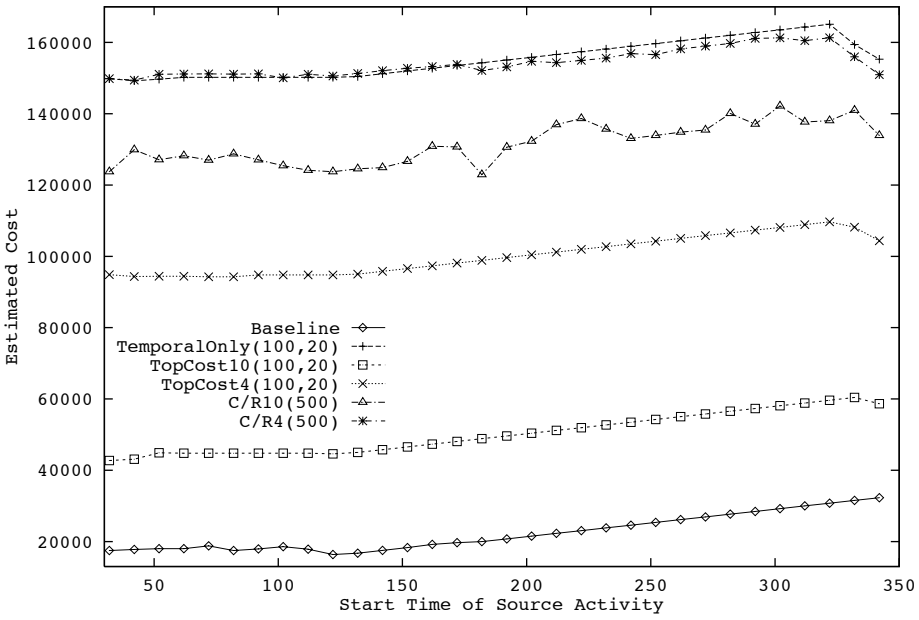

Figure 34. Raw Data for Problem 1, Problem Set 5

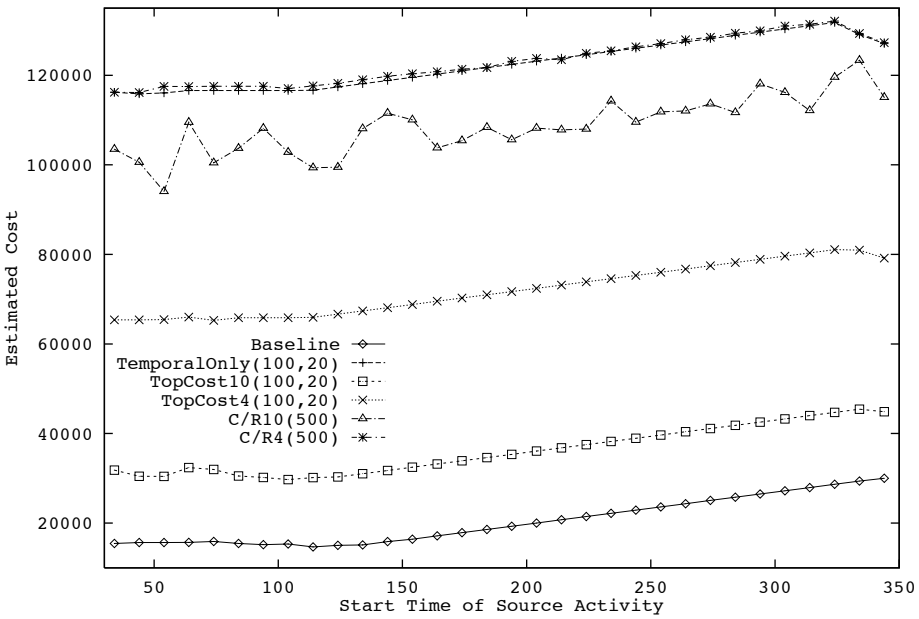

Figure 35. Raw Data for Problem 9 in Problem Set 8

A number of observations can be made from these data:

- Across all three problems, the TopCost plots seem to be close to a vertical translation of the Baseline plot. This indicates a high quality estimate. The Contention/Reliance plots are, in contrast, bumpy, indicating a less close matching with the baseline estimates
- In Figure 33, we see that the TopCost10 estimation is less than the baseline estimation. This occurs in other problems and in fact, in some problems, the TopCost4 estimations less than the baseline estimates. This arises from the fact, that in some cases, the TopCost algorithms are scheduling more of the non-zero cost activities than the Depth algorithm. This point calls into question the accuracy of the baseline estimations (see Section 5.7).

5.6.2 Texture Measurements

To compare the efficacy of the texture measurements we control for both the size of the subgraph and the settings of the parameters. This requires that we compare the algorithms on subgraphs of size 4 (the only size that the TemporalOnly algorithm uses) and use parameter settings of 1 and 500 for NumberOfSets and SetSize respectively (the only parameter settings that the Contention/Reliance algorithms use). The *EstGap* results are presented in Figure 36.

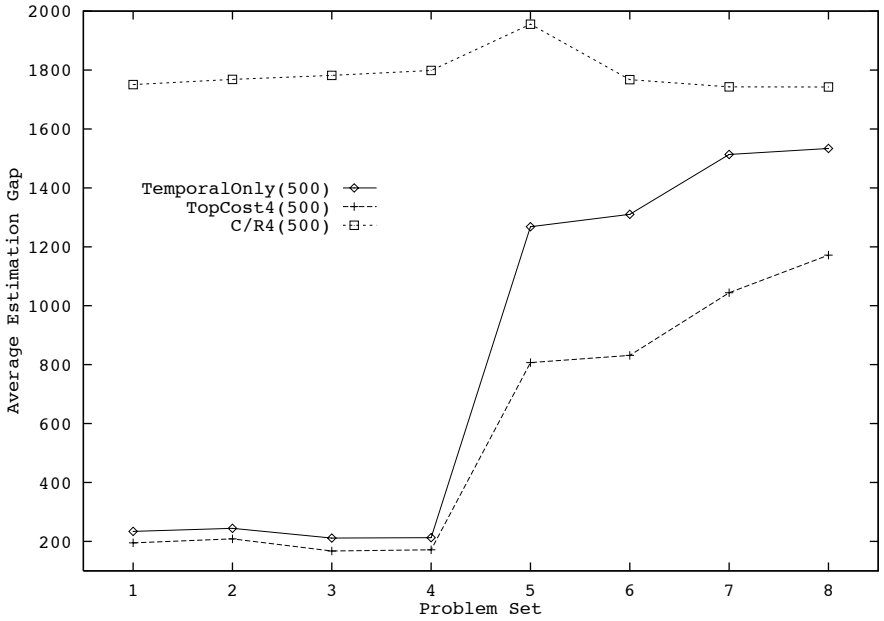

Figure 36. The *EstGap* Statistic for Estimations using Different Texture Measurement

Recall, that the *EstGap* statistic is a measure of the variation in the difference in estimates between the algorithm and the baseline. Observations:

- TopCost outperforms TemporalOnly which in turn outperforms the Contention/Reliance algorithm. While we expected that TopCost would produce superior estimates to Temporal-Only, it is surprising that Contention/Reliance algorithm performs so poorly. We will return to this point in our discussion.
- None of the algorithms perform well. A *EstGap* of 200 means that the standard deviation in the differences between the algorithm and the baseline estimates is 200. In other words, the estimations made by even the best algorithm estimates are not a constant difference from the baseline. This result contrasts to the raw data given in Figures 33 through 35. There, we noted that the TopCost algorithms appeared to be a vertical translation of the baseline. If this were the case the *EstGap* value would be 0. This conflict is an artifact of the scale used in Figures 33 through 35. The estimation costs are between 2000 and 160000. Given that scale, estimates with an *EstGap* measure of 200 does not appear to vary widely.

Figure 37 compares the *ChangeActivity* count for each of the algorithms. With the parameter settings used, each of algorithm finds a single solution using the backtracking algorithm and then 499 solutions by randomly changing the start time of one of the scheduled activities.

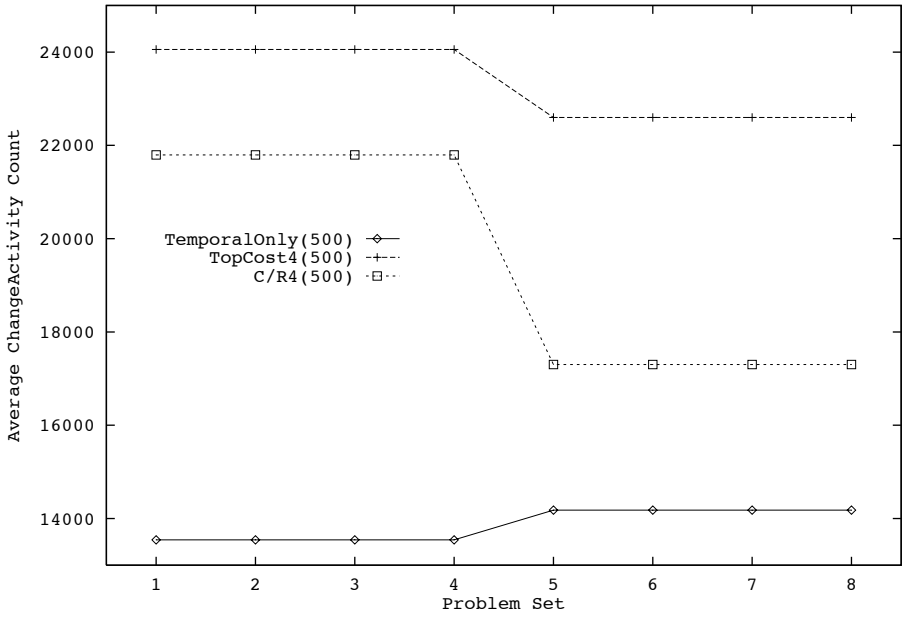

Figure 37. The *ChangeActivity* Count Estimations using Different Texture Measurement

Observation:

- We expect that the *ChangeActivity* count across the algorithms would be the same, since they are using the same sized subgraph and the same parameter settings. Figure 37 shows that this is not the case.

Further analysis reveals that the difference among the algorithms is in the number of assignments during the random phase of the algorithm. Normally, the random phase reassigns activities SetSize - 1 times. The only exception is when all the possible values of all the scheduled activities have been assigned or are inconsistent. The assignment of the source value significantly prunes the possible values of temporally connected activities. For example, the minimum make-span of a particular order is 100 time units, the earliest start time is 0, and the duration of each of the 5 activities is 20. The earliest temporally consistent start time for the final activity is 80. The final activity is the source activity and 80 is selected as a candidate value. After assigning the final task to 80, the value propagation *prunes the domains of the each of the temporally connected activities to a singleton set*. With the TemporalOnly algorithm, the temporally connected activities are the activities in the subgraph, therefore there is only a single consistent solution on the subgraph. Instead of generating 499 solutions, the random phase does not generate any because there are no consistent values to assign any of the activities in the subgraph.

This is seen to a lesser extent in the Contention/Reliance algorithm because pruning of values when marked activities are scheduled is lessened because the marked activities are not necessarily temporally connected and the Contention/Reliance assigns random values rather than the lowest consistent value.

5.6.3 Subgraph Size

The only estimation algorithms where the size of the subgraph is varied are the Contention/Reliance algorithm and the TopCost algorithm.[11] The *EstGap* statistics of both algorithms (with the NumberOfSets and SetSize parameters set at 1 and 500 respectively) are presented in Figure 38. The effect of the subgraph size on the *ChangeActivity* count is presented in Figure 39.

Two surprising results arise from the *EstGap* data in Figure 38:

- TopCost4 is as good as the TopCost10 algorithm on half of the problem sets. This indicates that a small subgraph of activities (e.g 4) were largely responsible for determining the impact of the scheduling decision.
- The Contention/Reliance algorithm on a larger subgraph actually performs *worse* than when it is executed on a small subgraph. This is unexpected, though it is mirrored in the variability seen in the C/R10 algorithm in the raw data (see Figures 33-35). We will return to this point in the discussion section below (see Section 5.7).

Figure 39 reflects the trend of a larger *ChangeActivity* count for the algorithms with larger subgraphs. This is expected since the size of the subgraph has significant effect of the number of assignments and, therefore, the number of times that value propagation is done in the constructive algorithm.

---

11. Recall that because the TemporalOnly algorithm formed its subgraph only via temporal constraint links, the size of the subgraph is a constant.

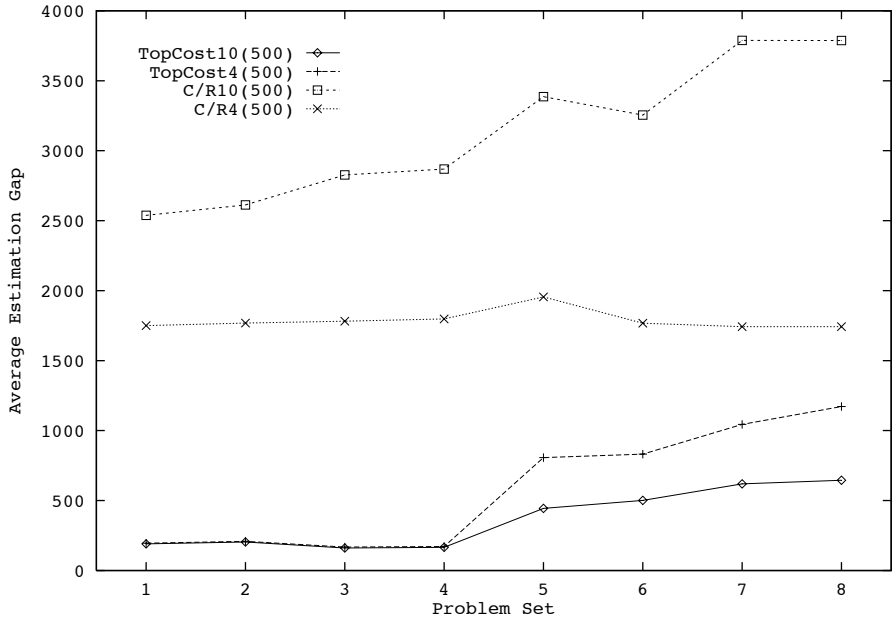

Figure 38. The Effect of Subgraph Size on the *EstGap* Statistic

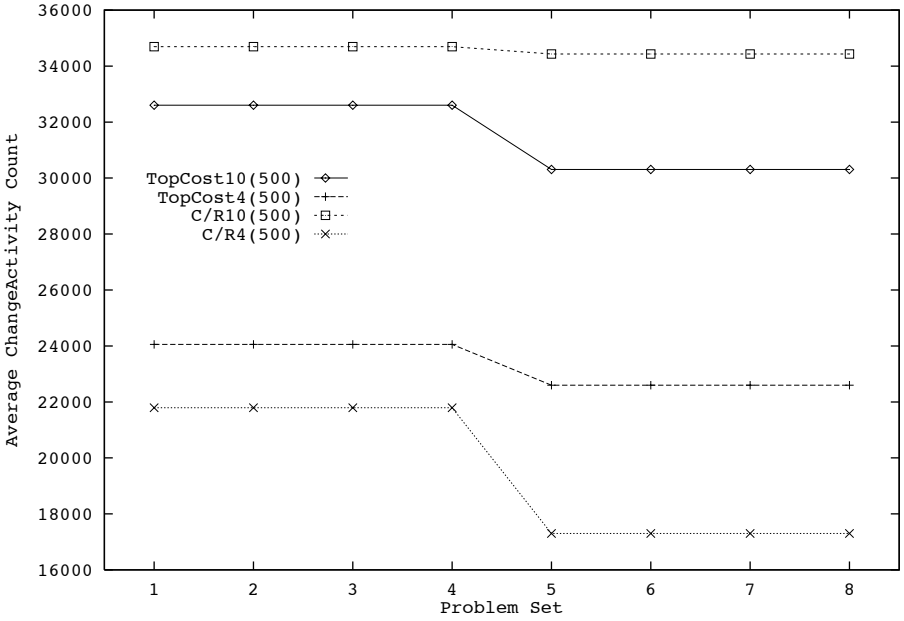

Figure 39. The Effect of Subgraph Size on the *ChangeActivity* Count

5.6.4 Parameter Settings

Two sets of parameter setting were used in the experiments. This first assigned NumberOfSets and SetSize to 5 and 20, respectively. These settings require that 5 solutions are produced for each value of the source activity by the backtracking algorithm. For each backtracking solution, 19 random solutions are found. The second set of parameters were 1 and 500 respectively for NumberOfSets and SetSize. The latter settings produce a single backtracking solution and 499 random solutions are for each value of the source activity.

The results in the *EstGap* and *ChangeActivity* measures are presented in Figures 40 and 41. Note that the different parameters, in the graphs, are specified by the SetSize and the total number of solutions. In cases where the two are equal, only one number is displayed.

Observations:

- Figure 40 shows that on 1 bottleneck problems, the different parameter settings have no effect on the *EstGap*. The algorithms that find fewer constructive solutions actually perform *better* on the problems with 2 bottlenecks than the corresponding algorithms that find a larger number of backtracking solutions.
- The effect on the *ChangeActivity* count is much as expected: the algorithms that use more constructive search (and hence more value propagation) incur a higher *ChangeActivity* count.

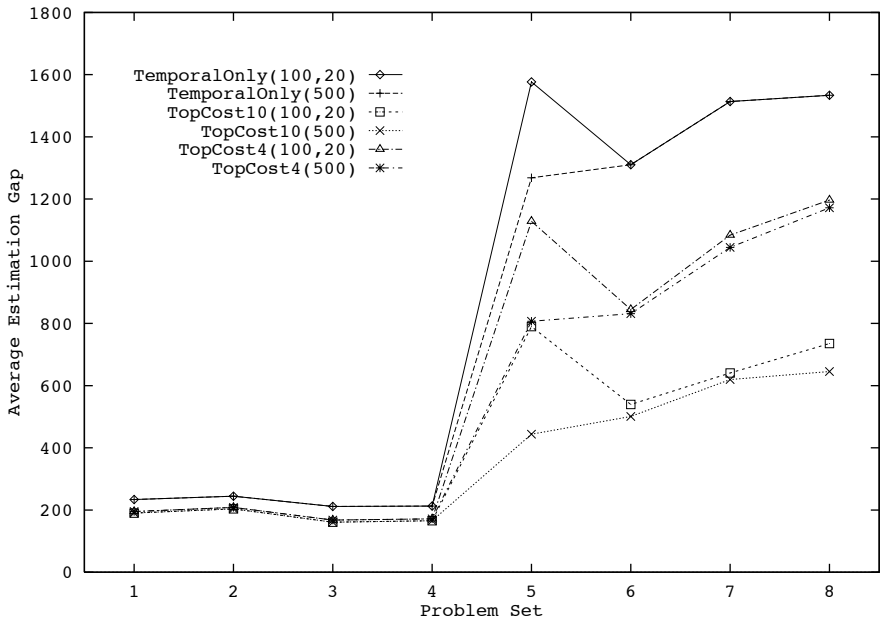

Figure 40. The Effect of Parameter Settings on the *EstGap* Statistic

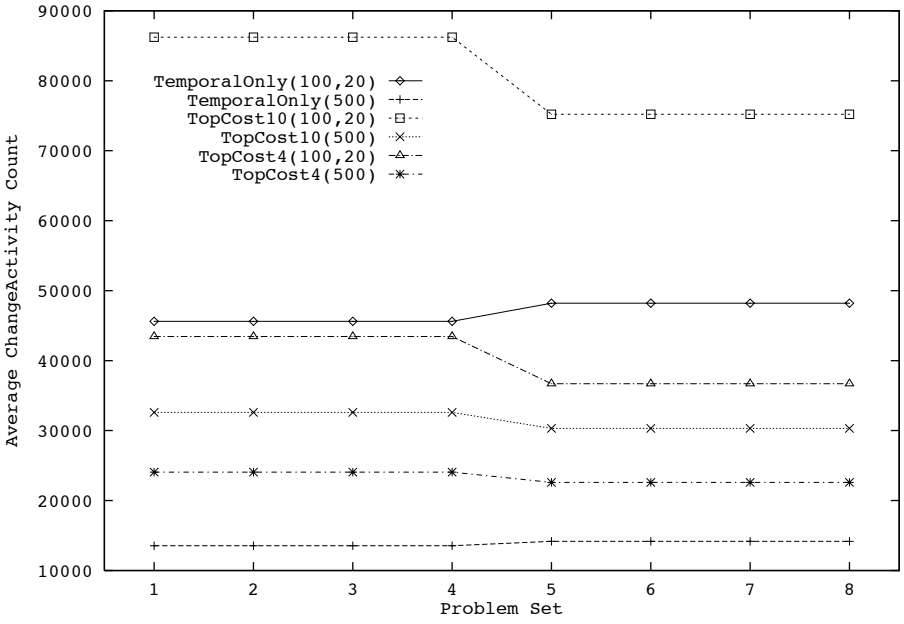

Figure 41. The Effect of Parameter Settings on the *ChangeActivity* Count

5.6.5 Overall

An overall picture of the performance of the various algorithms is given in Figures 42 and 43. These figures represent the performance of each algorithm using the parameters giving the best performance in terms of both *EstGap* and *ChangeActivity*. We define the best performance for the *EstGap* statistic to be lowest *EstGap* measure indicating the lowest standard deviation in the difference between the estimate and the baseline. For the *ChangeActivity* measure, the best performance is also the lowest. A low *ChangeActivity* value indicates less effort expended in forming the estimate.

With respect to the *EstGap* measure, Figure 42 shows that both TopCost algorithms perform better than any other algorithm. Further, the TemporalOnly algorithm outperforms the Contention/Reliance algorithms.

The *ChangeActivity* results are the lowest we observed for each algorithm. The TemporalOnly algorithm has the lowest *ChangeActivity* count while the algorithms using larger subgraphs have the higher counts.

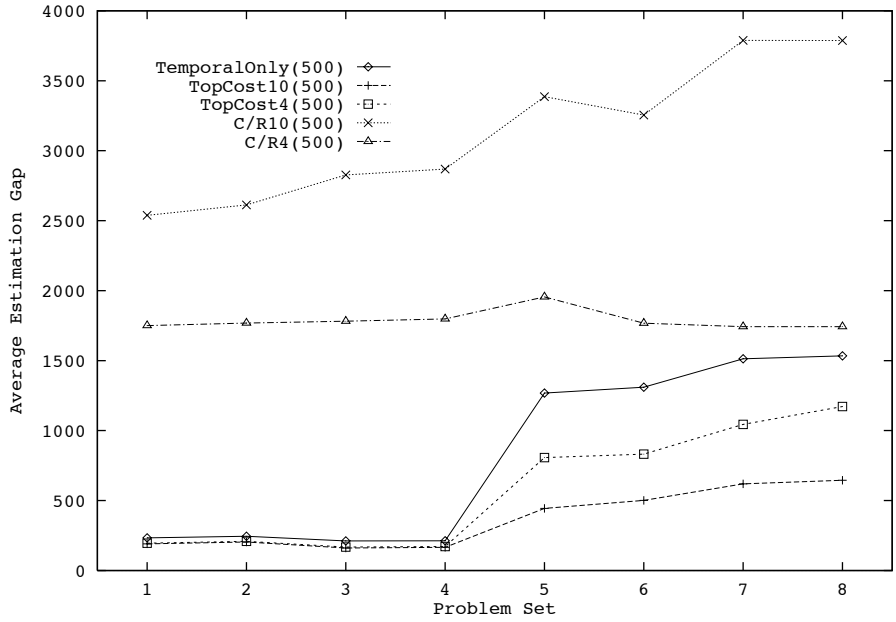

Figure 42. The Best *EstGap* Statistic for Each Texture Measurement and Subgraph Size

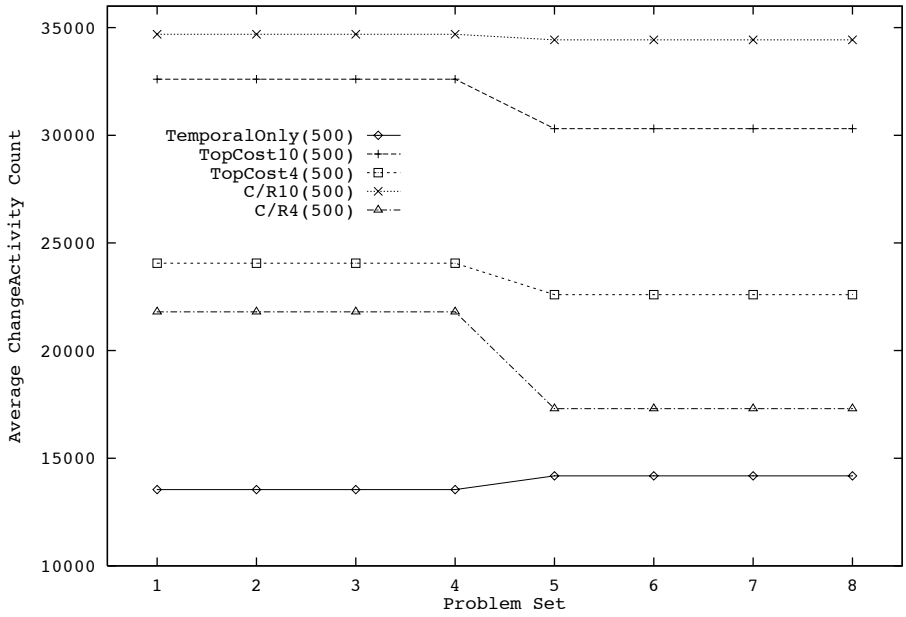

Figure 43. The Lowest *ChangeActivity* Counts for Each Texture Measurement and Subgraph Size

5.7 Discussion

In comparing our expectations and the experimental results, we find a number of surprising and anomalous results. In summary:

1. The raw data shows that on some problems the estimates found by TopCost10 and TopCost4 have a lower average cost than the baseline estimate (Figure 33).
2. The poor performance of the Contention/Reliance algorithms is demonstrated both by the *EstGap* statistic and in the bumpiness of the C/R plots in the raw data (Figures 33-36).
3. The estimates of the TopCost4 algorithm are as good as the TopCost10 estimates on the first 4 problem sets (Figure 38).
4. The estimates of the C/R10 algorithm are worse than those found by the C/R4 algorithm across all problems (Figure 38).

Recall that our expectations were based on the assumption that the baseline estimate was an accurate reflection of the actual impact of the scheduling decisions. Each of these points brings into question that assumption. We review the Depth algorithm that creates the baseline estimate and then look at each of the above points in turn.

5.7.1 The Depth Algorithm

The Depth algorithm uses a subgraph based on proximity to the source activity. All activities within a specified radius, L, are marked. The baseline estimate is calculated with $L = 1$. The structure of the experimental problems results in a subgraph of about 21 activities. Each of these activities is scheduled using the constructive algorithm and the cost of the solution is found. The cost for the Depth algorithm (as for all algorithms used here) is found by relaxing the constraints so that the assignments are satisfied. In cases where one or both of the activities are not assigned, a worst case estimate (largest cost) based on the remaining possible values for the unassigned activities is used.

It is important to note that the solution on the subgraph is not necessarily a partial schedule for the entire graph. It may be the case that, due to the assignments of the activities in the subgraph, no satisfying start time can be found for some activities outside the subgraph. Our cost estimates, then, are not necessarily based on partial, globally admissible solutions but rather on solutions for the subgraph. In general, if we could guarantee that our subgraph solutions were partial globally admissible solutions, we would be able to solve the whole scheduling problem. This caveat applies to all algorithms used here, not only the Depth algorithm.

5.7.2 TopCost vs. Baseline

The fact that a texture-based algorithm makes lower estimates than the baseline, indicates that there are relaxable constraints for which the Depth algorithm is using an higher estimate than the TopCost algorithm. Because we use the worst case estimate for relaxable constraints where one or both activities are not assigned, we expect algorithms that perform better analy-

sis (i.e. use fewer worst case estimates) will tend to find lower cost estimates. Further, we expect that the depth of analysis will correspond to the quality of the estimate: the more activities we assign, the more likely that the impact of a scheduling decision on the subgraph will reflect the global impact. Therefore, we expect that the lower estimates correspond to better quality estimates.

This reasoning must be tempered with the realization that the solutions on the subgraph are not necessarily partial solutions for the whole graph (see Section 5.3.5). It is possible that despite a lower raw cost estimate, the subgraph solutions are not actually part of an overall solution. We cannot therefore state categorically that a lower raw cost corresponds to increased accuracy. We believe, however, that this tends to be the case and so the lower raw costs of the TopCost algorithm in comparison with the baseline is evidence that TopCost outperforms Depth. The assumption of quality for the baseline estimate is, at the least, suspect given these results.

5.7.3 Poor Performance of Contention/Reliance

The Contention/Reliance algorithm is specifically designed to find the activities that rely heavily on a resource at a time when there is a high contention for the resource. It is the most sophisticated texture measurement used here and the most specific to the scheduling domain. Of any of the texture measurements, it gives the most attention to identification of activities on the basis of resource-related importance. The Contention/Reliance algorithm should provide the best estimate of the inter-order impact. A possible explanation for the poor performance of the C/R algorithms is that it is the baseline, rather than the C/R algorithms, that are at fault.

The results showing that the Contention/Reliance algorithm on a larger graph performs worse than on a smaller graph, further suggest that the baseline estimate is not accurate. If it is correct that the Contention/Reliance algorithm better reflects the impact via resource constraints, expanding the graph should find a better estimate of the inter-order costs. This is not what we found. A possible explanation is that the baseline is a poor estimate.

5.7.4 TopCost4 vs. TopCost10

By much the same argument used in the previous section, given an accurate baseline estimate, we expect that the TopCost10 algorithm would outperform TopCost4 on all problems. Our results show that on the first 4 problem sets, the two algorithms perform about the same, while on the latter 4 problem sets the TopCost10 algorithm performs better. We take the fact that the TopCost10 algorithm does not perform substantially better (across all problems) than the TopCost4 as a further indication that the baseline estimate does not accurately reflect the true situation.

5.7.5 Reassessing the Estimations

These results call into question the accuracy of the baseline estimates used in the experiments. It seems likely that the baseline estimate is inaccurate, however due to the complexity of solving the scheduling problem, no simple techniques are available to independently assess the quality of the baseline estimate.

We require an independent comparison of the subgraph identification algorithms. This comparison can be accomplished if the estimation algorithms are embedded within a complete scheduling system. Each of these algorithms can be used as the value ordering heuristic (while holding the other techniques within the scheduling algorithm constant). The texture measurement that more accurately reflects the actual impact will lead to better (e.g. lower cost) schedules in less time (due to the lessening of the backtracking effected by a good value ordering). We leave this integration for future work.

5.8 Conclusion

These experiments are the first demonstration, of which we are aware, that the propagation of cost information in a scheduling problem is not limited solely to temporal constraints.

We see these techniques as occupying a middle ground among the previous work in estimation of the impact of assignments. At one extreme is high-complexity, complete preference propagation [Sadeh 89] while at the other is the low-complexity, low-accuracy estimate of marginal cost propagation [Sadeh 91] . Therefore, in addition to introducing cost propagation across any types of constraints, this work integrates general heuristics to modulate the complexity of the propagation and the accuracy of the estimations. Both the generality of constraint propagation and the ability to integrate heuristics stems from the relaxation schema upon which these algorithms are built.

5.8.1 Building Practical Algorithms

Despite the generality of the techniques, the proposed propagation techniques are still impractical for use in a scheduling system. This is primarily due to the complexity of the search and the dependence on the ability to minimize backtracking. Specifically, to use these propagation methods in full-fledged scheduling a number of issues must be addressed:

- It is necessary to investigate efficient ways of updating the cost curves as scheduling decisions at other activities are made. The re-calculation of the impact of each value with each scheduling decision is clearly impractical.
- With large scheduling horizons, and therefore large domains for each activity, cost estimation for every possible value in the activity may be unnecessary. To minimize the time spent assessing likely values, it is desirable to filter values on the basis of inexpensive, fast, local criteria. In our experiments, we found a cost curve at a granularity of every tenth value. We suggested that further cost estimation could focus on values in the vicinity of the

lower cost values found by the coarse granularity algorithm. This is one possible approach. Further work is needed to identify and compare texture measurements that can be applied to such pre-filtering of values.

5.9 Summary

This chapter has presented an application of our relaxation schema to schedule optimization. We represent the sources of cost in a schedule by relaxable constraints and define a number of algorithms that estimate the impact of activity start times on the overall schedule. The estimates are based on the use of the costs of relaxations at activities within a subgraph as a texture measurement of tightness of constraints on those activities. These costs are propagated to the source activity where they are summed to form the estimate. The constraint relaxation schema allows propagation to flow over all types of constraints. The subgraph (upon which to base our impact estimates) can be identified without concern for the types of constraints.

The value impact estimates produced by a number of algorithms are compared to a baseline estimate. Results showed that all algorithms performed relatively poorly with respect to matching the baseline estimates. The number of anomalous results suggest that the baseline estimate for the impact of scheduling decisions does not accurately reflect the real impact. Further work is required, perhaps within a full scheduling system, to compare the texture measurements used to identify the subgraphs.

Chapter 6 Coordination of Multiple Agents

As noted in Chapter 1, our chief motivation for work in constraint relaxation is its application to coordination of multiple agents. We adopt a constraint-based model of a multiagent environment and present a sketch of a mediated constraint relaxation protocol for coordination. The protocol is demonstrated with the aid of an example from the domain of supply chain management. We conclude with a discussion of the bearing that the work in this dissertation has on coordination.

6.1 A Theory of Coordination

The study of the coordination has arisen from the recognition that a number of disciplines are concerned with domains in which autonomous agent work for individual or group goals in a shared environment [Malone 92] . These disciplines include such widespread examples as economics, biology, sociology, and computer science. The widely accepted definition for coordination is the *managing of dependencies between activities* [Malone 92] .

Within computer science much of the coordination work has been done in Distributed Artificial Intelligence [Distributed 87] [Distributed 89] . Previous work has addressed topics such as negotiation [Lesser 81] [Davis 83] [Rosenschein 85] [Durfee 87a] [Durfee 91] , distributed planning [Conry 86] [Pope 92] , and the modeling of other agents [Sycara 89] [Cohen 91] [Shoham 92] .

We are pursuing the development of a theory of coordination with immediate application to the area of supply chain management. Our model of coordination rests on the constraint-based problem solving model used throughout this dissertation. We view individual agents as constraint-based problem solvers. The agent interactions, therefore, are the creation, deletion, and exchange of constraints. Agents constrain each other through explicit commitments or through operations on the environment which indirectly place constraints on other agents. An example, of the former, in the supply chain would be for the manager of the resources to commit to the delivery of a raw material to a factory at a certain date. This commitment constrains the resource manager to make the delivery of the specified material and constrains the receiving factory in the actions it can take before and after delivery. An example of indirect interaction is when two factories use a shared resource pool. Use of the pool by one factory places constraints on the action of the other.

With this perspective, the environment is characterized by a distributed constraint graph shared among the agents. Coordination is necessary when an agent is unable to meet a commitment: the interdependencies between the activities at different agents require management. By failing to meet a commitment, the agent creates an infeasible constraint graph. Coordination is a reconfiguration of the graph, possibly including constraint relaxation, in order to re-establish feasibility.

6.2 A Mediated Approach to Coordination

In contrast to the negotiated approach taken by others (e.g. [Lesser 81] [Durfee 87a] [Durfee 87b] [Durfee 91]), we adopt a mediated approach to coordination. We identify a special agent, the mediator,[1] that has coordination knowledge and is responsible for re-establishing feasibility in the shared constraint graph.

6.2.1 The Role of the Mediator

When an event occurs that prevents an agent from meeting a constraint, the mediator will be notified. The mediator then builds an *augmented constraint graph* that represents, at some level of abstraction, the distributed constraint graph. This graph is built with information that the mediator has locally and from information gathered from other agents. The communicated information will typically be local constraints and relaxation costs that have an impact on the situation but are not yet represented in the augmented constraint graph. There is no need to represent the same level of detail in all parts of the graph. The mediator can represent important subgraphs to arbitrary detail by engaging in a dialogue with the relevant agents about constraints and the costs of various relaxations.

Once the augmented graph is formed, the mediator uses constraint relaxation to evaluate alternatives. It is likely that the forming of the constraint graph will be interleaved by evaluations of alternatives. The relaxation algorithm may identify a subgraph where more detail is required. The mediator can then engage in further dialogue with the relevant agents to flesh-out the appropriate areas of the graph. Once an acceptable solution has been found, the mediator makes suggestions to the agents involved as to the relaxations that will most efficiently re-establish feasibility.

6.2.2 Why Mediation?

Our adoption of a mediated approach is a starting point in our exploration of coordination. We intend to move toward more distributed methods base on inter-agent negotiation. There are a number of good reasons to adopt mediation, especially with our domain of application.

- Mediation minimizes the coordination knowledge required by each agent.

1. The agent may also have other functional abilities and responsibilities that are unconnected with meditation.

- The supply chain domain is highly-structured and only requires explicit coordination techniques when unexpected events occur.
- The constraint graph upon which relaxation needs to be performed is partially represented in the existing schedule or plan. The agent originally responsible for the scheduling or planning is a natural choice as a mediator.
- Coordination in the supply chain is necessary when the schedule is found to be infeasible. Scheduling difficulties can often be traced to a particular scarce resource [Smith 89] [Fox 90] [Sadeh 91] . The work of [Sathi 89] indicates that problems with such keystone elements can be solved better by mediated protocols.

We now illustrate our approach to coordination with a scenario from the supply chain.

6.3 Supply Chain Management Revisited

Recall that, in a manufacturing enterprise, the supply chain consists of all activities leading to the delivery of a product to a customer (e.g. marketing, material planning, production planning, production control, transportation, customer service). There are a number of agents within the supply chain that are responsible for these various activities.

At the logistics level, we view each factory within the enterprise as a resource, able to perform a number of activities. The activities produce a quantity of a particular resource over some time period and logistics-level scheduling assigns factories to produce specific quantities of resources at particular times. Factories commit to the high-level schedule and schedule the intra-factory activities to meet the commitment. As long as the operations of the enterprise do not violate the assumptions and aggregate information upon which commitments are based, no explicit coordination is necessary. When an unexpected event occurs such that an agent can not meet an existing constraint, the agents must coordinate their efforts to respond optimally.

6.3.1 An Example

In Chapter 1, we used a requested order change as an example of an unexpected, conflict-causing event. Here we present an instantiation of a supply chain simulation and show the actions taken by each agent in reaction to this event.

We use the following agents in our example:

- **Logistics**: responsible for the logistics-level scheduling.
- **Order Acquisition** (OA): responsible for all order-related contact with customers including order entry, cancellation, and modification.
- **Resource Manager** (RM): responsible for managing consumable resources (e.g. timely ordering of raw materials) and usable resources (e.g. scheduling of regular maintenance).
- **Factory Agents** (F_1, …, F_5): each of the five factories is represented, at the logistics level, by a single agent (typically the factory-level scheduler).

There are three scheduled orders each with a release date of t_0 and a due date of t_3. Each order consists of three activities with unit duration. The current set of orders and the schedule is in Table 7. The corresponding constraint graph is displayed it Figure 44. I should be noted that an activity at a factory may be an aggregate activity that the factory agent represents as a constraint graph in itself.

Start Times	**Factories**				
	F_1	**F_2**	**F_3**	**F_4**	**F_5**
t_0	A_{31}	A_{11}			A_{21}
t_1		A_{22}		A_{12}	A_{32}
t_2		A_{13}	A_{33}		A_{23}

TABLE 7. **Current Schedule in the Supply Chain Simulation**

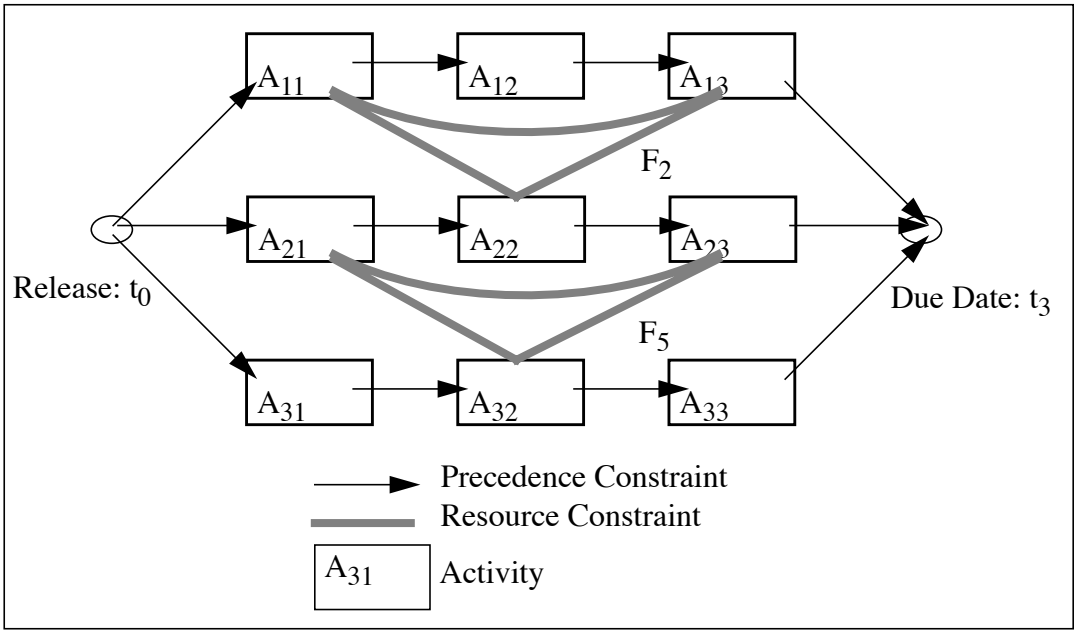

Figure 44. Logistics Level Constraint Graph of the Example

At time t_{-1}, the customer for O_3 requests that the order be changed requiring activity A_{33} to performed at factory F_2. The OA agent notifies Logistics of the customer's wishes and Logistics passes the change onto the factory. F_2 discovers that an infeasible situation ensues: the current schedule of activities does not allow F_2 to meet its commitments. Further, F_2 cannot reconfigure its local activities to compensate and so notifies Logistics. Logistics, operating as the mediator, investigates the following alternatives:

1. Can activities A_{11}, A_{22}, or A_{13} be moved to another factory?
2. Can delivery of some products be delayed?

In order to answer these questions, the mediator must request information from other agents. A representation of the communication that takes place for option 1 is as follows:

- Logistics to RM: "What factories can be perform activities A_{11}, A_{22} and A_{13}?"
- RM to Logistics: "F_4, but quality will decrease."[2]
- Logistics to OA: "What is the cost, due to relations with the customer, of the reduced quality on O_2 and O_1?"
- OA to Logistics: "3 each."

For option 2, Logistics must know the cost of late delivery of each order:

- Logistics to OA: "What is the cost of late delivery on each order?"
- OA to Logistics: "O_1: 5 per time unit, latest acceptable delivery is t_6."
- OA to Logistics: "O_2: 5 per time unit, latest acceptable delivery is t_6."
- OA to Logistics: "O_3: 2 per time unit, latest acceptable delivery is t_6."

This information from other agents allows the mediator to build the augmented constraint graph upon which a relaxation algorithm can then run. The algorithm will not necessarily choose one of the options, but rather it will investigate possibilities of a combination of moving some activities and of delaying delivery of some products.

In the example, the least expensive solution is to move A_{13} to F_4 and A_{33} to F_2 at a cost of 3. The schedule after relaxation is shown in Table 8. The corresponding constraint graph is displayed it Figure 45. Note the change in the resource constraints due to the relaxation.

Start Times	**Factories**				
	F_1	**F_2**	**F_3**	**F_4**	**F_5**
t_0	A_{31}	A_{11}			A_{21}
t_1		A_{22}		A_{12}	A_{32}
t_2		A_{33}		A_{13}	A_{23}

TABLE 8. **Schedule After Relaxation**

2. Work is progressing on a theory of quality that will make such a response meaningful [Kim 94] .

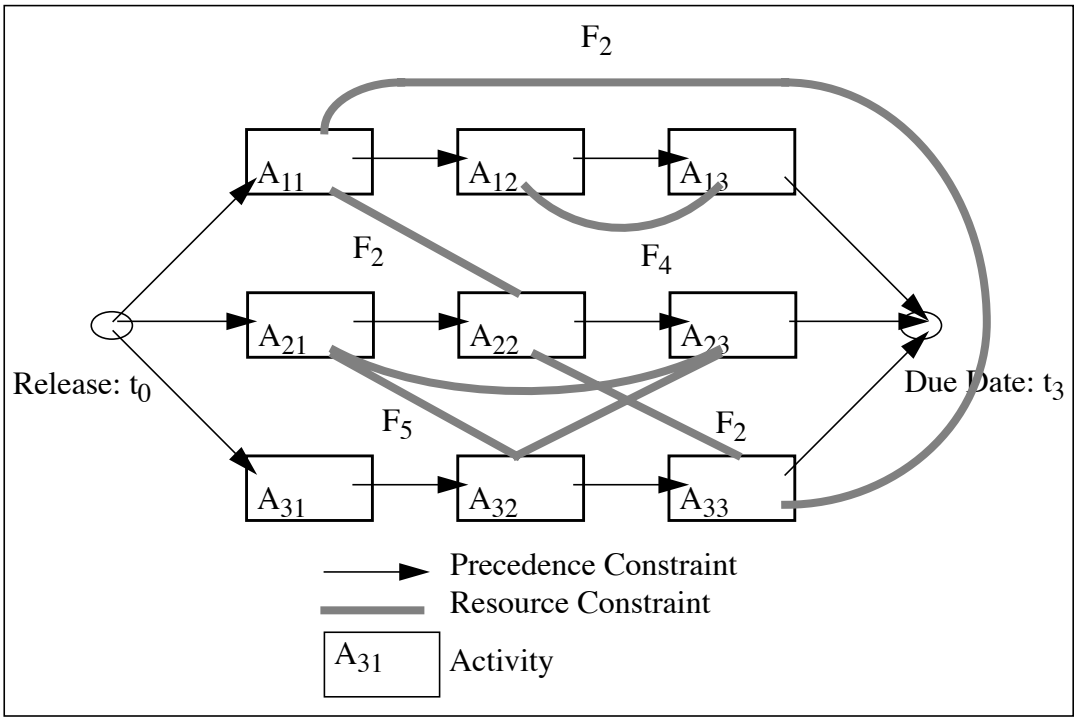

Figure 45. Logistics Level Constraint Graph After Relaxation

6.4 Conclusion

There are compelling parallels between the functionality of the mediator and the cost gathering algorithms described in Chapter 5. The cost gathering algorithm investigates the effect of assigning an activity to a particular start time. This investigation is carried out by the identification of a salient subgraph within the scheduling problem and the propagation of cost data through the graph. The assignment of a start time to a activity may have non-local effects and the cost gathering is an attempt to generate some estimate of those non-local effects. The estimates form a basis on which to choose a value with the least negative global impact.

The mediator has an infeasible constraint graph that is embedded within the distributed constraint graph, shared amongst the agents. The mediator must make modifications, such as reassigning some variables and/or relaxation of some constraint, in order to re-establish feasibility. The modifications that the mediator investigates also have non-local effects through constraints that may not be represented by the mediator. The mediator must estimate the non-local effects of relaxations, by the augmenting its constraint graph with further constraints and relaxation information, and choose a modification with the least negative global impact.

The constraint relaxation schema presented in this thesis is important for coordination because of its generality and ability to be integrated with heuristic decision making. The constraints that agents place on each other are not simply temporal constraints. The ability to apply con-

straint relaxation across all constraint types is a requirement for assessment of the relaxation decisions. Furthermore, there is no central representation of the constraint graph and, in any non-trivial agent environment, the graph will be too large to operate on directly. The ability to modulate search by heuristic decision making is a necessity even on the partial graph generated by the mediator.

Chapter 7 Concluding Remarks

7.1 Contributions

This thesis makes contributions in five areas:

- A generalization of the constraint model allowing representation of the semantic content of constraint relaxations.
- A flexible, general relaxation schema within which algorithms can be specified.
- A specification of three relaxation algorithms useful in small Partial Constraint Satisfaction Problems (PCSPs), constraint relaxation, and constraint satisfaction problems.
- Application of the propagation techniques and texture-based heuristics within the relaxation schema to schedule optimization.
- The modeling of multiagent coordination as a problem requiring constraint relaxation.

The final point was discussed in the previous chapter. We address the first four areas in turn.

7.1.1 A Generalized Constraint Model

The constraint model that we introduce uses a generator function at each constraint to produce alternative constraints and a cost function assessing the local impact of a constraint relaxation. This allows for very general approach to problem specification and the explicit use of relaxation generators as operators in constraint-directed search. The modification to the constraint model is a true generalization because it allows the use of either constraint satisfaction or constraint relaxation search techniques.

Our model expands the semantics of constraint relaxation because of the ability to assess the local cost of a specific relaxation. The impact of a relaxation depends not only on the original constraint, but also on the relaxation that is done. Simply ignoring a constraint has a very different effect than expanding the set of satisfying tuples by one entry. Previous work has failed to address this difference. The ability to represent and reason about the differential impact of modifications to problem constraints is a foundational requirement of any theory of constraint relaxation and of any relaxation-based search technique.

7.1.2 A Schema for Relaxation Algorithms

A schema for relaxation algorithms based on the propagation of various types of information between variables is presented. The schema defines a class of algorithms with varying complexity and performance characteristics. The specification of procedures used at five heuristic decision points is required:

1. Selection of the source variable from where propagation originates.
2. Selection of a set of candidate values for each variable.
3. Selection of a set of outgoing constraints from each variable.
4. Selection of a set of candidate constraints at each outgoing constraint.
5. Selection of the value-commitment at the source variable.

The schema allows these heuristic choices to be tailored to certain constraint types or specific run time conditions. By isolating the heuristics of an algorithm at a few key points, the schema allows for specification of a variety of algorithms of different complexity and performance. If algorithms are categorized by their complexity and performance characteristics, it may be possible to make the choice of algorithm at run time based on structural problem properties, time pressure, required solution quality, and other factors.

The central contributions of the relaxation schema are its generality and its ability to be integrated with arbitrarily complex heuristics. The generality allows the gathering of relaxation information across all types of constraints. This is demonstrated in Chapters 4 and 5, where we define algorithms based on the schema for PCSP graphs and schedule optimization graphs respectively. The integration of heuristics becomes extremely important when the size of the graphs is non-trivial. The results of the exponential algorithms for the PCSPs demonstrate that without heuristic decision making, graphs of 16 or more nodes are intractable.

7.1.3 Three Relaxation Algorithms for PCSPs

Using the relaxation schema, we specify three relaxation algorithms with the performance/complexity trade-off in mind: SMV, MMV, and MC. The latter algorithm is a variation of the MinConflicts repair algorithm [Minton 92] . Experiments compare performance against the partial extended forward checking algorithm, judged by [Freuder 92] to be the best PCSP algorithm investigated. Results showed that in almost all cases the SMV and MMV algorithms performed significantly fewer constraint checks than the PCSP algorithm while the quality of the solutions was close to optimal.

PCSPs are a special case of constraint relaxation, and, as such, they do not address the semantic issues that are focal in our constraint model. For this reason, we do not view the work on PCSPs in itself as having a large impact on constraint relaxation. We use PCSPs here as an example of the applicability and generality of the relaxation schema. More important is the indication, from the number of consistency checks, that the experimental problems are

approximately at the limit of size to which both the relaxation and complete algorithms can be applied. This underscores the requirement, noted above, for heuristic decision making in problems of non-trivial size.

7.1.4 Application of the Relaxation Schema to Schedule Optimization

We use the value and cost propagation of the relaxation schema to estimate the impact of assigning an activity to a start time in a schedule optimization domain. Our experiments to compare the quality of cost estimate produced by algorithms using three texture measurements were inconclusive due to the inability to find an accurate baseline estimate for the actual cost impact of scheduling decisions. We suggest that these estimation algorithms should be compared by integration of each algorithm within a larger scheduling system. The quality of the schedules and the time performance of the overall scheduler will then indicate the relative merits of the cost estimation algorithms.

Despite the lack of conclusive results, the texture measure-based algorithms represent the first example, of which we are aware, of the ability to dynamically choose a subgraph for cost estimation, independent of constraint type. We believe this ability is advantageous for two reasons:

1. The dynamic identification of the subgraph allows the modification of the basis for scheduling decisions as the schedule evolves. Furthermore, the size of the subgraph and depth of analysis on the subgraph can be tailored to both the quality and processing time requirements of the particular scheduling situations.
2. Texture measurements are a prime technique for the identification of this subgraph. Using dynamically calculated texture measurements as a basis for heuristic decisions can achieve significant algorithmic savings by limiting the size of the data set which must be searched over. These savings can come at little penalty to performance.

Our inability to form an accurate baseline estimate against which we could compare the algorithms, prevents us, in the work done here, from assessing either of these points.

7.2 Future Work

A described in Chapter 6, the main body of our future work will be in the application of constraint relaxation and the techniques developed here to coordination of multiple agents in a shared environment. Nevertheless, there are a number of other areas where interesting future research may be pursued. Four broad areas for future research are:

1. **The characterization of the trade-offs made by relaxation algorithms.** The relaxation algorithms are heuristic-based algorithms that are not guaranteed to find the optimal solution to problems. There is a classic trade-off of processing time for quality of solution. Given that this trade-off is made, it is necessary to characterize the types of solutions that

will not be found by algorithms defined within the relaxation schema. Such a characterization would allow the selection of algorithms based on the dynamic factors (e.g. time pressure, quality of solution desired by the user) surrounding a particular problem instance.

2. **The continued investigation of texture measurements as a basis for heuristic decisions.** The constraint research has, to this point, shown that exponential algorithms can only be avoided in very specific conditions. These conditions are rare in real problems and the effort required to create these conditions can be prohibitive. We require algorithms that make use of well-founded heuristic decisions to eliminate large portions of the search required to find a solution. Texture measurements provide this foundation.
3. **The exposition and investigation of relaxation as a general search technique.** We characterize relaxation throughout this thesis as changing the definition of a constraint. We believe that this can be used as a powerful general search technique that is complementary to constraint satisfaction search techniques.
4. **The application of the relaxation techniques to scheduling.** In characterizing the source of costs in a schedule as relaxable constraints, we achieve an increase in the scope of optimization criteria that can be expressed. In the scheduling domain, little work has been done on problems where a large number of constraints can be relaxed. Furthermore, research is also required on the efficient use of the general propagation techniques proposed on this dissertation. We believe that scheduling algorithms using such a technique will realize a significant increase in the quality of schedules produced.

7.3 Summary

Constraint relaxation is a general search method on any problem that can be expressed in the constraint paradigm. We have shown that a key its use is the recognition of the semantic nature of relaxation: the meaning of a constraint relaxation is embedded in the original meaning of the constraint and in the way in which the relaxation modifies the constraint.

We have presented a general algorithm schema for relaxation algorithms and shown its applicability to general constraint problems and schedule optimization. Future work will examine the application and further development of relaxation techniques in the area of multiagent coordination.

Bibliography

[Allen 83] Allen, J.F. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM. 26* (11):832-843, November, 1983.

[Bakker 93] Bakker, R.R., Dikker, F., Tempelman, F., and Wognum, P.M. Diagnosing and Solving Over-determined Constraint Satisfaction Problems. *Proceedings of IJCAI-93*. 1993.

[Bazaraa 90] Bazaraa, M.S., Jarvis, J.J., and Sherali, H.D. *Linear Programming and Network Flows.* John Wiley & Sons, Inc., New York, 1990.

[Borning 87] Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., and Woolf, M. Constraint Hierarchies. *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*. 1987.

[Borning 88] Borning, A., Maher, M., Martindale, A., and Wilson, M. *Constraint Hierarchies and Logic Programming.* Technical Report 88-11-10, Computer Science Department, University of Washington, November, 1988.

[Cohen 91] Cohen, P.R. and Levesque, H.J. Teamwork. *Nous 35.* 1991.

[Conry 91] Conry, S.E., Kuwabara, K., and Lesser, V.R. Multistage Negotiation for Distributed Constraint Satisfaction. *IEEE Transactions on Systems, Man, and Cybernetics. SMC-21* (6):1462-1477, 1991.

[Conry 86] Conry, S.E., Meyer, R.A., Lesser, V.R. *Multistage Negotiation in Distributed Planning.* Technical Report 86-67, Department of Computer and Information Science, University of Massachusetts, December, 1986.

[Cooper 92] Cooper, M. *Visual Occlusion and the Interpretation of Ambiguous Pictures.* Ellis Horwood, Limited, Chichester, West Sussex, England, 1992.

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

[Davis 94] Davis, E.D. *ODO: A Constraint-Based Scheduler Founded on a UNified Problem Solving Model.* Technical Report TR-EIL-94-1, Enterprise Integration Laboratories, Department of Industrial Engineering, University of Toronto, 1994.

[Davis 83] Davis, R. and Smith, R.G. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence. 20* (1):63-109, 1983.

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

[Dechter 89] Dechter, R. and Pearl, J. Tree Clustering for Constraint Networks. *Artificial Intelligence*. *38* 353-366, 1989.

[Dechter 90] Dechter, R., Dechter, A., and Pearl, J. Optimization in Constraint Networks. *Influence Diagrams, Belief Nets, and Decision Analysis*. In Oliver, R.M. and Smith, J.Q., John Wiley and Sons, Ltd, Chicester, England, 1990.

[Descotte 85] Descotte, Y. and Latombe, J.C. Making Compromises among Antagonist Constraints in a Planner. *Artificial Intelligence. 27* 183-217, 1985.

[Distributed 87] Huhns, M.N. (editor). Volume 1 *Distributed Artificial Intelligence*. Pitman Publishing & Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[Distributed 89] Gasser, L. and Huhns, M.N. (editor). Volume 2 *Distributed Artificial Intelligence.* Pitman Publishing & Morgan Kaufmann Publishers, San Mateo, CA, 1989.

[Dorn 92] Dorn, J., Slany, W., and Stary, C. Uncertainty Management by Relaxation of Conflicting Constraints in Production Process Scheduling. *Working Notes AAAI Spring Symposium on Practical Approaches to Scheduling and Planning*. 1992.

[Dubois 93] Dubois, D., Fargier, H., and Prade, H. The Use of Fuzzy Constraints in Job-Shop Scheduling. *Working Notes of the IJCAI-93 Workshop on Knowledge-Based Production Planning, Scheduling, and Control*. 1993.

[Durfee 87a] Durfee, E.H. and Lesser, V.R. Using Partial Global Plans to Coordinate Distributed Problem Solvers. *Proceedings of IJCAI-87*, pages 875-883. 1987.

[Durfee 87b] Durfee, E.H., Lesser, V.R., and Corkill, D.D. Cooperation Through Communication in a Distributed Problem Solving Network. Volume 1. *Distributed Artificial Intelligence.* In Huhns, M.N., Pitman Publishing & Morgan Kaufmann Publishers, 1987, pages 29-58, Chapter 2.

[Durfee 91] Durfee, E.H. and Montgomery, T.A. Coordination as Distributed Search in a Hierarchical Behavior Space. *IEEE Transactions on Systems, Man, and Cybernetics. SMC-21* (6):1361-1378, 1991.

[Fox 86] Fox, M.S. Observations on the Role of Constraints in Problem Solving. *Proceedings of the Sixth Canadian Conference on Artificial Intelligence*. 1986.

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

[Fox 89] Fox, M.S., Sadeh, N., and Baykan, C. Constrained Heuristic Search. *Proceedings of IJCAI-89*. 1989.

[Fox 90a] Fox, M.S. and Sadeh, N. Why Is Scheduling Difficult? A CSP Perspective. *Proceedings of the Ninth European Conference on Artificial Intelligence*. 1990.

[Fox 90b] Fox, M.S. Constraint-Guided Scheduling - A Short History of Research at CMU. *Computers in Industry. 14* 79-88, 1990.

[Fox 92] Fox, M.S. *Integrated Supply Chain Management*. Technical Report, Enterprise Integration Laboratories, Department of Industrial Engineering, University of Toronto, April, 1992.

[Freuder 89] Freuder, E. Partial Constraint Satisfaction. *Proceedings of IJCAI-89*. 1989.

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

[Freuder 85] Freuder, E. A Sufficient Condition for Backtrack Bounded Search. *Journal of the ACM*. *32* (4):755-761, 1985.

[Garey 79] Garey, M.R. *Computers and Intractability*. W.H. Freeman and Company, San Francisco, 1979.

[Gaschnig 77] Gaschnig, J. A General Backtrack Algorithm that Eliminates Most Redundant Tests. *Proceedings of IJCAI-77*. 1977.

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

[Ghedira 94] Ghedira, K. Partial Constraint Satisfaction with Multi-Agent Systems Combined with Simulated Annealing Process. *Proceedings of 14th International Conference on Artificial Intelligence, Expert Systems, and Natural Language, Avignon-94*. 1994.

[Gruninger 94] Gruninger, M., and Fox, M.S. *Proceedings of Second International Conference On Cooperative Information Systems*. 1994. To Appear.

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

[Hillier 80] Hillier, F.S. and Lieberman, G.J. *Introduction to Operations Research.* Holden-Day, Inc., San Francisco, 1980.

[Huhns 91] Huhns, M.N. and Bridgeland, D.M. Multiagent Truth Maintenance. *IEEE Transactions on Systems, Man, and Cybernetics. SMC-21* (6):1437-1445, November/December, 1991.

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

[Interrante 93] Interrante, L.D. and Rochowiak, D.M. A Distributed Agent Architecture for Integrating AI and OR in the Dynamic Scheduling of Manufacturing Systems. *Working Notes of the IJCAI-93 Workshop on Knowledge-Based Production Planning, Scheduling, and Control*. 1993.

[Johnston 92] Johnston, M.D. Spike: AI Scheduling for Hubble Space Telescope After 18 Months of Orbital Operations. *Working Notes AAAI Spring Symposium on Practical Approaches to Scheduling and Planning*. 1992.

[Kim 94] Kim, H.K. and Fox, M.S. Formal Models of Quality and ISO 9000 Compliance: An Information Systems Approach. *48th Annual Quality Congress of the American Society of Quality Control*, pages 17-23. 1994.

[Kirkpatrick 83] Kirkpatrick, S., Gelarr, Jr., C.D., and Vecchi, M.P. Optimization by Simulated Annealing. *Science*. *220* (4598):671-680, May, 1983.

[Kumar 92] Kumar, V. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*. *13* (1):32-44, 1992.

[Lansky 88] Lansky, A.L. Localized Event-Based Reasoning for Multiagent Domains. *Computational Intelligence. 4* 319-340, 1988.

[Lesser 81] Lesser, V.R and Corkill, D.D. Functionally Accurate, Cooperative Distributed Systems. *IEEE Transactions on Systems, Man, and Cybernetics. SMC-11* (1):81-96, January, 1981.

[Lowry 92] Lowry, M. Integration of AI and OR Technologies: Issues and Tradeoffs. *Working Notes AAAI Workshop on Production Planning, Scheduling, and Control*. 1992.

[Mackworth 77] Mackworth, A. Consistency in Networks of Relations. *Artificial Intelligence*. *8* 99-118, 1977.

[Malone 92] Malone T.W., and Crowston, K. *The Interdisciplinary Study of Coordination.* Technical Report, Center for Coordination Science, Massachusetts Institute of Technology, December, 1992.

[Mcmahon 92] McMahon, M.B. and Dean, J. A Simulated Annealing Approach to Schedule Optimization for the SES Facility. *Working Notes AAAI Spring Symposium on Practical Approaches to Scheduling and Planning*. 1992.

[Minton 92] Minton, S., Johnston, M., Philips, A., and Laird, P. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*. *58* 161-205, 1992.

[Mittal 90] Mittal, S. and Falkenhainer, B. Dynamic Constraint Satisfaction Problem. *Proceedings of AAAI-90*. 1990.

[Morton 86] Morton, T.E., Lawrence, S.R., Rajoagopalan, S., and Kekre, S. *MRP-STAR: PATRIARCH's Planning Module.* Technical Report, Graduate School of Industrial Administration, Carnegie Mellon University, December, 1986.

[Nadel 88] Nadel, B.A. Tree Search and Arc Consistency in Constraint Satisfaction Algorithms. *Search in Artificial Intelligence.* In L. Kanal and V. Kumar, Springer-Verlag, New York, 1988.

[Nagel 91] Nagel, R.N. et al. *21st Century Manufacturing Enterprise Strategy: An Industry Led View.* Technical Report, Iacocca Institute, Lehigh University, Bethlehem PA, 1991.

[Navinchandra 87] Navinchandra, D. and Marks, D.H. Design Exploration Through Constraint Relaxation. *Expert Systems in Computer-Aided Design.* In J. Gero, Elsevier Science Publishers B.V., 1987.

[Navinchandra 91] Navinchandra, D. *Exploration and Innovation in Design.* Springer-Verlag, New York, 1991.

[Nuitjen 93] Nuitjen, W.P.M., Aarts, E.H.L., van Erp Taalman Kip, D.A.A., and van Hee, K.M. Randomized Constraint Satisfaction for Job-Shop Scheduling. *Working Notes of the IJCAI-93 Workshop on Knowledge-Based Production Planning, Scheduling, and Control*. 1993.

[Page 91] Page, C.D. Jr. and Frisch, A.M. Generalizing Atoms in Constraint Logic. *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*. 1991.

[Phillips 76] Phillips, D.T., Ravindran, A., and Solberg, J.J. *Operations Research.* John Wiley & Sons, Inc., New York, 1976.

[Pope 92] Pope, R.P., Conry, S.E., and Meyer, R.A. Distributing the Planning Process in a Dynamic Environment. *Working Papers of the 11th International Workshop on Distributed Artificial Intelligence*. 1992.

[Ravindran 87] Ravindran, A., Phillips, D.T., and Solberg, J.J. *Operations Research Principles and Practice.* John Wiley & Sons, Inc., New York, 1987.

[Rosenfeld 76] Rosenfeld, A., Hummel, R.A., and Zucker, S.W. Scene Labeling by Relaxation Operations. *IEEE Transactions on Systems, Man, and Cybernetics. 6* 1976.

[Rosenschein 85] Rosenschein, J.S. and Genesereth, M.R. Deals Among Rational Agents. *Proceedings of IJCAI-85*, pages 91-99. 1985.

[Sadeh 89] Sadeh, N. and Fox, M.S. *Preference Propagation in Temporal/Capacity Constraint Graphs.* Technical Report CMU-RI-TR-89-2, The Robotics Institute, Carnegie Mellon University, January, 1989.

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

[Sadeh 94] Sadeh, N. *Micro-Opportunistic Scheduling: The MICRO-BOSS Factory Scheduler.* Technical Report, The Robotics Institute, Carnegie Mellon University, 1994.

[Sathi 89] Sathi, A. and Fox, M.S. Constraint-Directed Negotiation of Resource Reallocations. Volume 2. *Distributed Artificial Intelligence.* In Michael N. Huhns and Les Gasser, Pitman Publishing & Morgan Kaufmann Publishers, 1989, pages 163-193, Chapter 8.

[Selman 92] Selman, B., Levesque, H., and Mitchell, D. A New Method for Solving Hard Satisfiability Problems. *Proceedings of AAAI-92*. 1992.

[Shapiro 81] Shapiro, L.G. and Haralick, R.M. Structural Descriptions and Inexact Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence. 3* (11):504-519, September, 1981.

[Shoham 92] Shoham, Y. and Tennenholtz, M. On the Synthesis of Useful Social Laws for Artificial Agent Societies. *Proceedings of the National Conference on Artificial Intelligence*, pages 276-281. 1992.

[Smith 89] Smith, S.F., Ow, P.S., Matthys, D.C., and Potvin, J.Y. OPIS: An Opportunistic Factory Scheduling System. *Proceedings of International Symposium for Computer Scientists*. 1989.

[Stallman 77] Stallman, R.M. and Sussman G.J. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence*. *9* 135-196, 1977.

[Sycara 89] Sycara, K.P. Argumentation: Planning Other Agents' Plans. *Proceedings of IJCAI-89*, pages 517-523. 1989.

[Van 89] Van Hentenryck, P. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

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

[Yokoo 92a] Yokoo, M. and Durfee, E.H. Distributed Search Formalisms for Distributed Problem Solving: Overview. *Proceedings of the 11th International Workshop on Distributed Artificial Intelligence*. 1992.

[Yokoo 92b] Yokoo, M., Durfee, E.H., Ishida, T., and Kuwabara, K. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. *Proceedings of the 12th International Conference on Distributed Computing Systems*. 1992.

[Yokoo 93] Yokoo, M. Constraint Relaxation in Distributed Constraint Satisfaction Problems. *Proceedings of the 5th International Conference on Tools with Artificial Intelligence*. 1993.

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

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