University of London
Imperial College of Science, Technology and Medicine
Centre for Planning and Resource Control (IC-Parc)

Improving Backtrack Search: Three Case Studies of Localized Dynamic Hybridization

Hani El-Sakkout

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Engineering of the University of London and the Diploma of Imperial College, June 1999

To my families, old and new.

Abstract

Hybrid constraint programming algorithms can outperform their pure counterparts on constraint satisfaction and optimization problems. This thesis considers how hybridization can be used to improve the conventional backtrack search algorithm. Specifically, techniques for *localized*, *dynamic* hybridization are proposed and investigated: the methods hybridized with backtrack search target specific sub-problems, and co-operate with backtrack search as it executes.

The thesis consists of three case studies exploring different problems. Each case study describes an original algorithm that uses this style of hybridization to enhance the efficiency of backtrack search. A conclusion reviews the forms of hybridization the algorithms exemplify.

Acknowledgements

I would like to thank all those who have aided the writing of this thesis, starting with my first supervisor Mark Wallace for many detailed and helpful technical discussions which have contributed greatly to the quality of the work; and my second supervisor Barry Richards who generously gave me guidance and support when it was needed.

I also wish to thank all my many friends and colleagues at IC-Parc: Tom Richards and Stefano Novello for taking on extra project burdens while I was writing up; Vassilis Liatsos and Joachim Schimpf for our lively and fun technical discussions; Christiane Morgan and Lee Flanagan for everything from parking permits to wedding cakes; Zoe Leech and Lee McLoughlin for exposing several dark system mysteries; and everyone else for countless acts of help. I would also like to give thanks to my other friends, particularly Amin El-Kholy for his invaluable advice and moral support; Simon Taylor for helping me see the funny side; and Farid Ajili for enjoyable late-night debates.

Finally, I want to express my utmost gratitude to my wife Carmen, my mother, my father, and my brother for their advice, love, and encouragement, without which this thesis would not have happened.

Contents

INTRODUCTION 16

1 Overview **17**

1.1 General Motivation for Hybridization 17

1.2 Thesis Context & Contributions 19

1.2.1 Thesis Context 19

1.2.2 Thesis Contributions 20

1.3 Document Outline 21

2 Problems and Methods **23**

2.1 Problem Definitions 23

2.1.1 Constraint Satisfaction 24

2.1.2 Optimization 25

2.1.3 Relating Optimization to Satisfaction 26

2.1.4 Notation & Terminology 26

2.2 Features of CP Search Algorithms 28

2.2.1 Search in CP Algorithms 28

2.2.2 Soundness and Completeness 29

2.2.3 Search Algorithm Categories 30

2.3 Backtrack Search 31

2.3.1 Description of the Algorithm 32

2.3.2 The Stages of Backtrack Search 33

2.3.3 Optimization 34

2.3.4 Extensions 35

2.4 Alternative Search Strategies 39

2.4.1 Local Search 40

2.4.2 Dual Assignment Search 43

2.4.3 Mathematical Programming Methods 44

2.5 Summary 46

3 The Hybridization Context **47**

3.1 The Hybridization Conditions 47

3.2 Support Methods for Backtrack Search 50

3.3 Hybridization Issues 51

3.3.1 Selection & Scoping 52

3.3.2 Synchronization & Integration 53

3.4 Remainder of the Thesis 54

CASE STUDY I **55**

4 Adaptive Propagation **56**

4.1 Introduction 56

4.1.1 Overview 56

4.1.2 Background 57

4.1.3 Case Study Outline 59

4.2 Arc Consistency and Forward Checking 60

4.2.1 Basic Notation 60

4.2.2 FC- and AC-monitoring 61

4.2.3 When is AC More Effective than FC? 61

4.2.4 AC-superiority 63

4.3 Adaptive Arc Propagation 65

4.3.1 AAP Initialization 66

4.3.2 AAP Maintenance 67

4.4 Anti-Functional Reduction 69

4.4.1 AFR Initialization 70

4.4.2 AFR Maintenance and Propagation 71

4.5 Empirical Results 72

4.5.1 Experimental Setup 72

4.5.2 Results 74

4.6 Conclusion 77

CASE STUDY II **79**

5 Sub-Problem Delegation **80**

5.1 Introduction 80

5.1.1 Overview 80

5.1.2 Fleet Assignment Background 82

5.1.3 Outline of the Case Study 84

5.2 The Fleet Assignment Problem 85

5.2.1 Description 85

5.2.2 Definition 87

5.3 Representing Fleet Assignment as a CSOP 90

5.4 CP Backtrack Search (BT) 91

5.4.1 Overview of Algorithm 92

5.4.2 Enhancing the Algorithm 92

5.4.3 Backtrack Algorithm Pitfalls 95

5.5 The Hybridization Technique 97

5.5.1 Sub-Problem Delegation — the Hybridization Form 98

5.5.2 The Form Specialized for Fleet Assignment 99

5.5.3 The Hybridization Pseudo-Code 101

5.5.4 Summary 101

5.6 The Forward Search Methods 103

5.6.1 Limited Discrepancy Search 103

5.6.2 Forward Search by LDS 104

5.6.3 Forward Search by Hill-Climbing LDS 105

5.7 A Comparison of the Algorithms 107

5.8 Related Work 113

5.9 Conclusion 116

CASE STUDY III 118

6 Probe Backtracking 119

6.1 Introduction 119

6.1.1 Overview 120

6.1.2 Outline of the Case Study 121

6.2 Motivation for Dynamic Scheduling 122

6.2.1 A Toy Example 122

6.2.2 Commercial Application 123

6.3 Problem Definitions 125

6.3.1 The Minimal Perturbation Problem 125

6.3.2 Introduction to the KRFP 126

6.3.3 Activities in the KRFP 127

6.3.4 Resources in the KRFP 129

6.3.5 Temporal Constraints in the KRFP 129

6.3.6 Definition of the KRFP 130

6.4 Representing the KRFP as a CSP 131

6.5 A Minimal Perturbation Variant of the KRFP 132

6.6 Properties of the Temporal Sub-Problem 134

6.6.1 Minimal Perturbation in the Temporal Sub-problem 135

6.6.2 Transforming All Schedule Violations into Resource Constraint Violations 139

6.7 A Suite of Algorithms for Minimal Temporal Perturbation 140

6.7.1 The Constraint Backtrack Algorithms (CB) 142

6.7.2 The Probe Backtrack Algorithms (PB) 146

6.8 Performance Comparisons 153

6.9 Conclusion 157

6.9.1 Discussion 157

6.9.2 Related Work 161

6.9.3 Summary 163

CONCLUSION **164**

7 Review of Hybridization **165**

7.1 Form I: Adaptive Propagation 166

7.1.1 Description 166

7.1.2 Hybridization Issues 166

7.1.3 Potential and Limitations 167

7.2 Form II: Sub-Problem Delegation 168

7.2.1 Description 168

7.2.2 Hybridization Issues 169

7.2.3 Potential and Limitations 170

7.3 Form III: Probe Backtracking 172

7.3.1 Description 172

7.3.2 Hybridization Issues 173

7.3.3 Potential and Limitations 175

7.4 Future Research 176

8 Summary of Contributions **179**

References **182**

List of Figures

2.1 backtrack: The chronological backtrack algorithm 33

4.1 AC-superiority vs. support-complement ($|X| = 10$) 65

4.2 AAP-Initialization 67

4.3 AAP-Maintenance 68

4.4 AAP propagation procedure 68

4.5 AFR method list initialization 71

4.6 AFR scoping list maintenance 72

4.7 Average time (25,000 trials) versus no. of variables 74

4.8 Average time (25,000 trials) versus domain size 74

4.9 Average time (10,000 trials) versus P_{density} 75

4.10 Average time (10,000 trials) versus $P_{\text{applicability}}$ 76

4.11 Average time (10,000 trials) versus $P_{\text{constriction}}$ 76

4.12 Percentage improvement in time/operations versus average backtracks 77

5.1 An example of a fleet assignment problem 86

5.2 An example of the hybridization form 98

5.3 fleet_hybrid: A hybridization technique for fleet assignment 102

5.4 Optimization performance on **P-1** 109

5.5 Optimization performance on **P-2** 110

5.6 Optimization performance on **P-3** 111

5.7 Optimization performance on **P-4** 112

6.1 An example of a schedule modified to reduce resources 122

6.2 constraint_backtrack (CB): A procedure for the resource feasibility phase 143

6.3 probe_backtrack (PB): An improved, probe backtracking algorithm for the resource feasibility phase 147

6.4 Time/spaceout percentages for 20 & 30 task problems (problem set 1) 155

6.5 MIP time/spaceouts versus resource reduction and max. time shift (problem set 3) 158

6.6 CB+LP+Edge time/spaceouts versus resource reduction and max. time shift (problem set 3) 159

6.7 PB(Unimod)+Edge+Alt time/spaceouts versus resource reduction and max. time shift (problem set 3) 160

7.1 Search management: the future for backtrack search? 178

List of Tables

4.1 An example relation matrix (describing the constraint $X \leq Y$) 62

5.1 The fleet assignment problems 107

5.2 The fleet assignment algorithms 107

5.3 Percentages of flights changed in best solutions 113

6.1 Problem set 1: Search results 155

6.2 Problem set 1: MIP vs. PB(Unimod), on problems solved to optimality by both 156

6.3 Problem set 2: Search results 156

INTRODUCTION

Chapter 1

Overview

1.1 General Motivation for Hybridization

Effective algorithms for constraint satisfaction and optimization often involve the hybridization of several methods. This thesis studies aspects of algorithm hybridization from the perspective of the constraint programming (CP) paradigm.

Constraint satisfaction and optimization problems are commercially critical yet notoriously difficult to solve. Production and transportation scheduling, risk management, and communication network design are just a few examples of these ubiquitous but challenging problems. The principal obstacle to their solution is their underlying combinatorial complexity. They are usually NP-hard, and therefore it is very unlikely that general polynomial-time algorithms will ever be found to solve them.

Fortunately, the efficiency of some algorithms can be improved by specializing them for the problem at hand; most problems have characteristics that make reductions in computational effort possible. Constraint programming (CP) is an emerging paradigm that facilitates the construction and specialization of such algorithms.

CP is distinct from other paradigms in that its research has focused strongly on the development of techniques for the satisfaction of symbolic and disjunctive constraints. These are often the source of problem complexity. A number of powerful CP methods already exist to aid the solution of problems with such constraints.[1] The fact that these methods are simple and versatile greatly assists the process of algorithm specialization.[2]

Nevertheless, the process of algorithm specialization remains a daunting task, even for CP experts. It can be particularly complicated for problems that approach the scale and complexity of real-world requirements. These problems are referred to in the literature as large-scale combinatorial optimization problems (LSCOs). Three basic features co-exist in a typical LSCO:

1. **Problem Structure.**
   An LSCO typically comprises several sub-problems, each having very different characteristics.

2. **Large Size.**
   The sizes of the input data are large.

3. **Problem Specificity.**
   Every LSCO is unique. This is because of the diversity of detail in real-world problems, and because even very similar LSCOs are differentiated by the relative sizes of their sub-problems, and by differences among the constraints that relate them.

[1] Witness for example the increasingly sophisticated local consistency checking methods.

[2] Many researchers have compared CP with alternative paradigms on specific problems with varying results (the comparison most often made is with operations research (OR) approaches rooted in mathematical programming). However, the early case studies of the CHIP team (notably Van Hentenryck et al. in [91, 90]) support the by now prevalent view that CP method specialization is an especially effective approach for the development of constraint satisfaction and optimization algorithms, both in terms of development times, and algorithm quality.

Viewed in combination, these three features underscore the need for hybridization. Typically, the *structure* and *scale* of LSCOs means that no single algorithm (or algorithm configuration) perfectly addresses all their parts. Consequently, many effective algorithms for solving them involve the hybridization of several specialized algorithms. Moreover, their *specificity* means that previously developed algorithms are rarely applicable again without extension or modification. Thus, by their very nature, LSCOs place hybridization at the top of the CP research agenda.

1.2 Thesis Context & Contributions

1.2.1 Thesis Context

This thesis researches CP hybridization via the construction of hybrid algorithms for three problems, two of which are LSCOs.

Arguably, the eventual goal of research into hybridization is to automate the hybridization process. However, the selection and combination of techniques to handle a specific problem is currently still a "black art" — human experts mainly depend on their intuition and on rules of thumb built from past experience.

Thus most research into hybridization is directed at two intermediate objectives. The first is to broaden knowledge of hybridization by *case studies*, which investigate the creation of new hybrid algorithms for solving specific problems. Since the area of CP hybridization is not yet mature, this is where hybridization research is predominantly focused. The second objective is to extract from these hybrid algorithms wider *hybridization forms*: the term "hybridization form" describes a hybridization technique that is general enough to integrate more than just one set of methods.

The thesis pursues the first objective via three different case studies, and explores

the second objective in the thesis conclusion. All the algorithms developed sit within one hybridization context:

1. They consist of a *backtrack search* procedure controlling and co-ordinating search, with a set of hybridized *support methods* assisting it.

2. These support methods are *localized*: they are brought in to assist backtrack search on specific sub-problems. If necessary, sub-problems may overlap, and/or change as the search progresses.

3. The support methods are *dynamically hybridized*: they co-operate with backtrack search during its execution. They are not integrated as simple preprocessing steps before (or post-processing steps after) the execution of backtrack search.

4. The hybrid algorithms are *sound* and *complete* — with one small caveat.

1.2.2 Thesis Contributions

In the thesis, each of the three case studies presents original hybrid algorithms, and an empirical study that compares them with alternative algorithms. While the hybrid algorithms presented are very different from each other, they all satisfy the four hybridization conditions, as stated above.

The problems addressed are as follows:

- The first algorithm applies a hybridization form denoted *adaptive propagation*, and solves *binary constraint satisfaction* problems. While these problems are clearly not LSCOs, they provide a test-bed for the exploration of this particular form of hybridization. In the case study, two consistency methods, forward checking and arc-consistency, are automatically and dynamically hybridized

to support better the backtrack search. The exercise raises general issues concerning the automatic and dynamic hybridization of consistency methods.

- The second algorithm applies a hybridization form named *sub-problem delegation.* It addresses a commercial airline LSCO, *fleet assignment.* Two variants of the algorithm hybridize repair-based search methods with backtrack search, to improve the quality of solutions and the overall performance of the backtrack algorithm. The hybridization form demonstrates how an independent search procedure can enhance the performance of backtrack search, by controlling some of the search decisions.

- The third algorithm utilizes another form, *probe backtracking*, and solves a wide class of *minimal perturbation scheduling* LSCOs. This form also integrates the backtrack search procedure with an independent search technique, in this case a Simplex-based linear optimizer. However, the form is different from that of the second case study, using instead the independent search method to generate "probe" or "tentative" assignments. These are used at run-time to guide the backtrack search, and reduce the size of the problem it searches.

1.3 Document Outline

The thesis is divided into five parts: the introduction, the three case studies, and the conclusion.

- In addition to the present chapter, the introductory part includes Chapters 2 & 3. The former introduces the principal methods used in the thesis, and the latter raises issues concerning the construction of hybrid algorithms within our hybridization context.

- Case Study I describes the adaptive propagation algorithm for binary con-

straint satisfaction. The performance of the first hybrid algorithm is studied empirically, by comparison with algorithms applying only one type of consistency checking, on random binary constraint satisfaction benchmarks.

- Case Study II explores the sub-problem delegation approach on fleet assignment. An empirical study compares the hybrids presented with conventional backtrack search, on four real airline problems.

- Case Study III, detailing how probe backtracking may be used for minimal perturbation scheduling, is the largest and the deepest of all three. The third hybrid algorithm is presented, and comparisons made with a suite of alternative search methods. Experimental results on randomly generated problems show that this hybridization form can greatly improve the behaviour of backtrack search algorithms.

- Finally, the hybridization forms utilized in the three case studies are reviewed in Chapter 7, before Chapter 8 concludes the thesis with a summary of contributions.

Chapter 2

Problems and Methods

This chapter introduces the constraint satisfaction and optimization problems, and the principal algorithms and methods hybridized in the case studies.

2.1 Problem Definitions

Today, constraint satisfaction is the focus of an important area of artificial intelligence (AI) research. Its roots lie in image processing [97, 66] but its wider importance was soon recognized [60, 33, 43], eventually leading to the emergence of constraint programming (CP) as a distinct field of research. A number of introductory texts exist: in particular, Tsang surveys the key constraint satisfaction concepts and methods [88], while Marriott et al. describe CP languages and practice [62].

Most CP research is focused on solving instances of its two core problems, namely the constraint satisfaction problem (CSP) and the constraint satisfaction and optimization problem (CSOP). In this thesis, since the problems solved all involve *finite domains*, two particular specializations of the CSP and the CSOP — respec-

tively the finite CSP (FCSP) and the finite CSOP (FCSOP) — are directly relevant. Definitions for these problems are related below, together with relevant terminology.

2.1.1 Constraint Satisfaction

The basic constraint satisfaction problem is defined as follows:

Definition 2.1.1 *Constraint Satisfaction Problem (CSP)*
*A **constraint satisfaction problem** is a triple (V, D, C), where:*

- *V is a set of n variables $\{v_1, \ldots, v_n\}$;*
- *D is a function mapping each variable in V to a set of objects of arbitrary type:*

$$D : V \rightarrow \textit{set of objects (of any type)}$$

  *Let d_i denote the set of objects mapped from $v_i \in V$ by D. The objects are denoted the **values** of v_i, and the set d_i the **domain** of v_i.*

- *and C is a set of m relations (constraints) on subsets of the variables in V.*

*A **solution** to a CSP (V, D, C) is an assignment that maps every variable $v_i \in V$ to a value from its domain d_i such that every constraint in C holds.*

As mentioned in Chapter 1, constraint satisfaction problems are computationally difficult to solve. In fact, answering the question whether a solution exists for a CSP is in general an NP-complete decision problem [66].

The finite constraint satisfaction problem (FCSP) [61] is a class of CSPs where the domains are finite sets:

Definition 2.1.2 *Finite Constraint Satisfaction Problem (FCSP)*
*A **finite constraint satisfaction problem** is a CSP (V, D, C) where every domain d_i of a variable $v_i \in V$ is a finite set of values.*

2.1.2 Optimization

In real-world problems there are often preferences that cannot be expressed as constraints in the CSP model, since they are not strictly necessary in a solution. In such situations *optimization functions* are used to evaluate the quality of solutions. *Optimization algorithms* are applied to obtain solutions that are of good quality. We relate a definition below that extends the CSP to include a single optimization function [88]. The extended problem and its finite domain variant are general enough to capture most LSCOs, and will be used to model the two tackled in this thesis.[1]

Definition 2.1.3 *Constraint Satisfaction & Optimization Problem (CSOP)*
*A **constraint satisfaction and optimization problem** is a tuple (V, D, C, g):*

- *(V, D, C) is a CSP whose set of solutions is S;*
- *and g is an optimization function that maps every solution $\sigma \in S$ to a numerical value $g(\sigma)$ representing the solution's cost.*

*A **solution** to the CSOP is a solution to the CSP (V, D, C). An **optimal solution** σ^* to the CSOP minimizes $g(\sigma^*)$:*[2]

$$\forall \sigma \in S : g(\sigma^*) \leq g(\sigma)$$

[1] However LSCOs with several optimization criteria may be expressed better as problems with multiple optimization functions.

[2] In an alternative interpretation, the function g measures solution's value, and $g(\sigma^*)$ should be maximized.

Definition 2.1.4 *Finite Constraint Satisfaction & Optimization Problem (FCSOP)*
A ***finite constraint satisfaction and optimization problem*** *is a CSOP* (V, D, C, g) *where every domain* d_i *of a variable* $v_i \in V$ *is a finite set of values.*

2.1.3 Relating Optimization to Satisfaction

Clearly, the CSP and the CSOP are very closely related. Specifically: an optimal solution σ^* to a CSOP (V, D, C, g) satisfies $g(\sigma^*) \leq B$, where B is some numeric value, *if and only if,* there exists a solution σ to the CSP (V, D, C'), where C' includes all the constraints in C plus an additional constraint enforcing $g(\sigma) \leq B$. This means that algorithms designed for constraint satisfaction can also be used for optimization. For integral valued optimization functions, the bounds on $g(\sigma^*)$ can be narrowed until its value is found, then the set of optimal solutions to the CSOP is the solution set of the CSP extended with the additional constraint $g(\sigma) = g(\sigma^*)$.

2.1.4 Notation & Terminology

The definitions of terms relating to the CSP have been placed together below for easy reference. The terms will be used later to describe problem features and algorithm behaviour.

Let (V, D, C) describe a CSP. The following notation is used in the definitions:

- $\alpha, \beta, \gamma, \cdots$ represent assignments mapping subsets of the variables of V to values from their corresponding domains.
- V_α denotes the subset of V assigned by assignment α.
- $V \downarrow c$ denotes the subset of V related by a constraint $c \in C$.

Constraints

- A **unary constraint** has arity 1, a **binary constraint** has arity 2, while more generally an **n-ary constraint** has arity n.

Assignments

- The **length** of α is the number of variables in V_α.
- A **partial assignment** is any assignment of length greater than zero.
- α is a **complete assignment** iff $V_\alpha = V$.
- α is said to **extend** β **by** γ iff : $V_\alpha = V_\beta \cup V_\gamma$ and $V_\beta \cap V_\gamma = \emptyset$; the values assigned by α to the variables in V_β are identical to those assigned by β; and lastly the values assigned by α to the variables in V_γ are identical to those assigned by γ. α is also described as an **extension** of β and γ, and β and γ are described as **sub-assignments** of α.

Constraint satisfaction and violation

- An assignment α **satisfies** $c \in C$ iff it assigns all the variables in $V \downarrow c$ and c holds under the values assigned by α.
- An assignment α **violates** $c \in C$ iff it assigns all the variables in $V \downarrow c$ and c does not hold under α.
- An assignment α is said to be **consistent** iff no constraint in C is violated by α, and **inconsistent** otherwise.

The CSP

- The **solution set** of a CSP is the set of its solutions.

- A CSP is **solvable** if it has at least one solution, and **infeasible** otherwise.
- A **binary CSP** contains only unary and binary constraints.
- The **constraint hyper-graph** of a CSP is the undirected hyper-graph {N,E} where the nodes of N correspond to the variables of V ($N = V$), and the hyper-edges of E correspond to the constraints in C ($E = \{V \downarrow c | c \in C\}$).

2.2 Features of CP Search Algorithms

Diverse search algorithms exist for constraint satisfaction and optimization. However, all such algorithms share some basic features. We informally introduce these before discussing specific search methods.

2.2.1 Search in CP Algorithms

The core of any constraint satisfaction (or optimization) search algorithm is its **search strategy**: this describes the internal algorithm states that constitute "search nodes", and the "search ordering" method that is used to move among them.[3]

A **search node** is the internal algorithmic state that is assessed by the algorithm before it decides whether to terminate, or to move to another search node. It is usual when solving FCSPs (but not necessary) that each search node is associated with a particular partial or complete assignment. An algorithm moves from one search node to another until it terminates. The set of all search nodes potentially reachable by the algorithm whilst solving a particular problem is known as its **search space**. A subset of the search space, the **solution space**, is the set of search nodes at which

[3]The reader should be aware that the terms defined here are widely used, often with different meanings. The definitions given here are to clarify later discussions.

the algorithm outputs a solution. Thus the search nodes — and consequently the search and solution spaces — are dependent both on the problem and the algorithm solving it.

The **search ordering** method is the procedure that is invoked by the algorithm at one search node to obtain the next search node. The algorithm uses the search ordering method to direct its movement among the search nodes. Notice that if the search ordering method leads to a solution early in the search, much of the search space will remain unvisited by the algorithm: according to the above definitions, the search and solution spaces are independent of the search ordering method.

2.2.2 Soundness and Completeness

The notions of **soundness** and **completeness** describe two fundamental properties of a search algorithm, and are defined, *for constraint satisfaction* as follows:

Definition 2.2.1 *Soundness (Satisfaction Algorithms)*
A constraint satisfaction algorithm is ***sound*** *if and only if every assignment it outputs is a solution.*

Definition 2.2.2 *Completeness (Satisfaction Algorithms)*
A constraint satisfaction algorithm is ***complete*** *if and only if the algorithm guarantees, in a finite number of steps, either to generate a solution, or to prove that no solutions exist.*

Thus, for constraint satisfaction, in a sound algorithm the solution space for the problem addressed is always a (non-strict) subset of the problem's solution set, while in a complete algorithm the converse holds. In sound and complete constraint satisfaction algorithms the solution space and the solution set are identical.

We go on to define soundness and completeness *for constraint satisfaction & optimization* algorithms. Notice that, in the strictest sense, the goal of an optimization algorithm is to obtain the optimal solution. The definitions are made to reflect this:

Definition 2.2.3 *Soundness (Satisfaction & Optimization Algorithms)*
A constraint satisfaction & optimization algorithm is ***sound*** *if and only if every assignment it outputs is an optimal solution.*

Definition 2.2.4 *Completeness (Satisfaction & Optimization Algorithms)*
A constraint satisfaction & optimization algorithm is ***complete*** *if and only if the algorithm guarantees, in a finite number of steps, either to generate an optimal solution, or to prove that no solutions exist.*

Since the algorithms presented in this thesis are constraint satisfaction algorithms that have been extended for optimization, an algorithm could be sound and/or complete in the satisfaction sense without being so in the optimization sense. To eliminate confusion, we distinguish two different levels soundness & completeness: **sat-soundness** & **sat-completeness**, and **opt-soundness** & **opt-completeness**.

2.2.3 Search Algorithm Categories

In the thesis, all algorithms from the CP paradigm are divided into three categories, based on their search strategy:

Backtrack Search

Backtrack search algorithms implement various forms of depth-first tree search. In constraint satisfaction, backtracking algorithms normally associate each search node with a partial assignment that is potentially part of a solution:

they progress search by instantiating variables (effectively adding a constraint fixing an unassigned variable to a value from its domain). However, backtrack search could also be conducted by the addition of other constraints.[4]

Local Search

Local search methods (also referred to as *repair-based* algorithms) start the search by making a set of possibly contradictory assumptions describing a solution, and then progress by changing them until they are consistent with each other. In constraint satisfaction, the assumptions usually take the form of a complete, inconsistent assignment to the problem variables. Search moves make incremental changes to this assignment until it is transformed into a consistent solution.

Dual Assignment Search

This is a category created here to include algorithms that associate a search node with *both* a partial, consistent assignment, *and* a complete, possibly inconsistent assignment.

Backtrack search is described in the next section. Section 2.4 then considers the other search strategies encountered in the thesis, introducing both local and dual assignment search, as well as the basic mathematical programming methods.

2.3 Backtrack Search

Chapter 1 revealed that, in accord with the chosen hybridization context, the hybrid algorithms of the thesis are all based on backtrack search, with other methods acting in support. This section describes the details of the basic backtrack search

[4]For a simple example, see the description of domain splitting and its illustration on a real application in [89]. Later, the algorithms of Case Study III will supply another example of how search can progress through the addition and retraction of constraints.

algorithm, before the following section introduces other search strategies that will later be hybridized, or used for comparison.

2.3.1 Description of the Algorithm

The basic backtrack search algorithm presented is the FCSP variant of **chronological backtracking** [41], denoted backtrack. This operates on a single partial assignment that is incrementally extended if it is found to be consistent, and shortened whenever it is not. backtrack starts at the root search node with no variables assigned. It moves to a new search node by selecting a single unassigned variable and assigning it a value from its domain. If it is found immediately or subsequently that the resulting partial assignment cannot be extended to a solution, this variable assignment is undone. This step is known as a **backtrack**. Instead, the variable is assigned another untried value from its domain, or, if none exist (i.e. the current partial assignment cannot be extended at all) the preceding variable assignment is undone. Search is terminated when a solution is found or when the absence of solutions is proved by a backtrack to the root search node.

Figure 2.1 gives the pseudo-code for chronological backtracking. The algorithm is initialized by setting Assignment to be an empty assignment; OtherVarDoms to be the set of variable and domain pairs for the (initially unassigned) problem variables; and Constr to be the set of problem constraints. The first step is to call consistency_check. In the basic chronological backtracking algorithm, this simply checks that Assignment does not violate any of the constraints in Constr.[5] If any constraints are violated, consistency_check returns $\perp$ indicating that the assignment is inconsistent and a backtrack must occur — otherwise, the algorithm continues, attempting to extend the assignment towards a solution. Unless there are no more unassigned variables (in which case Assignment is a solution) the function select_var chooses and

[5]It will later be extended to return modified domains and constraints.

```
1.  begin backtrack(Assignment, OtherVarDoms, Constrs)
2.      Result := consistency_check(Assignment, OtherVarDoms, Constrs)
3.      if Result == ⊥ then
4.          return ⊥
5.      if OtherVarDoms == ∅ then
6.          return Assignment
7.      (v_i, d_i) := select_var(OtherVarDoms)
8.      OtherVarDoms':= OtherVarDoms −{(v_i, d_i)}
9.      d'_i:= d_i
10.     while d'_i ≠ ∅ do
11.         val := select_val(d'_i)
12.         Assignment':= extend_assignment(Assignment, v_i, val)
13.         Returned:= backtrack(Assignment', OtherVarDoms', Constrs)
14.         if Returned ≠⊥ then
15.             return Returned
16.         d'_i := d'_i - {val}
17.     return ⊥                        /* "Assignment" cannot be extended */
18. end backtrack
```

Figure 2.1: backtrack: The chronological backtrack algorithm

returns an unassigned variable and its domain. The algorithm will attempt to assign this variable. A "while loop" selects untried values from the domain of the variable, one at a time: d'_i represents the value in d_i untried so far. The function select_val uses heuristics to choose the best value from d'_i. The procedure appropriately updates OtherVarDoms and Assignment to assign the variable to the selected value before recursing. If Assignment' has been recursively extended to a solution, this solution is returned, otherwise alternative values from d'_i are attempted. $\perp$ is returned by the algorithm if none are successful, to report that Assignment cannot be extended.

2.3.2 The Stages of Backtrack Search

The chronological backtrack algorithm is characterized by three separate stages: these are denoted here the **consistency**, **branching** and **decision** stages. The

hybrid algorithms presented later will investigate hybridization issues in each of the three stages.

The consistency stage. At each level of recursion, the procedure consistency_check performs consistency checking, reporting any constraint violations. This ensures algorithm *sat-soundness*: the algorithm only outputs complete assignments that have been shown by consistency_check to satisfy all the constraints.

The branching stage. At this stage select_var partitions the remaining search problem. By selecting a variable v_i to assign, select_var partitions the problem into k sub-problems, where k is the cardinality of the associated domain d_i. In each sub-problem, the chosen variable is assigned a different value from its domain. Clearly, each sub-problem can be searched separately. Algorithm *sat-completeness* is respected because the k sub-problems contain all the initial problem's solutions, i.e. the union of the solution sets of these sub-problems is equal to the solution set of the unpartitioned problem.

The decision stage. The sub-problems are searched systematically and exhaustively in a recursive process, until a solution is found or problem infeasibility is proved. In the decision stage, select_val chooses a value, thus selecting a specific sub-problem from the set of unvisited sub-problems. On backtracking, the next best value is chosen, and so forth, until a solution is found or no more choices are possible.

2.3.3 Optimization

The close relationship between satisfaction and optimization means backtrack can also be applied to optimization problems, as explained in Section 2.1.3. The process

known as **branch and bound** [55] — one of the most well known mechanisms for extending satisfaction algorithms for optimization — will be applied in the optimization algorithms of Case Studies II and III. It involves the incorporation of new constraints into the constraint set, to force new solutions to be better than ones previously found.

Note that, with the appropriate branch and bound procedure, *the sat-sound and sat-complete backtrack algorithm can be extended to an opt-sound and opt-complete algorithm if necessary.*

2.3.4 Extensions

The chronological backtracking algorithm backtrack is applied widely due to its simplicity as well as its benign space complexity — $O(nd)$ where n is the number of FCSP variables and d is the size of the largest domain. However, the algorithm's exponential worst-case time complexity ($O(d^n c)$ where c is the number of constraints) is also reflected in its average-case behaviour.

Extensions to the basic algorithm are required for satisfactory average-case performance. There are three principal approaches to enhancing the performance of backtrack. First, sophisticated *search ordering* heuristics can be used to increase the chances that a solution is found early. Second, at each search node, *local consistency* methods may be deployed to identify values in the domains of unassigned variables that lead to an inconsistent extension of the present partial assignment. Third, *learning and intelligent backtracking* strategies record information across backtracks to avoid repeatedly searching the same sub-problems.

In Section 3.2, we supplement these approaches with a new category, **forward search**, corresponding to the hybridization of methods that independently search parts of the problem faced by the backtrack algorithm. The benefits of forward

search will be seen in Case Studies II & III. For the time being though, only the first three enhancement strategies are discussed.

Search Ordering for Backtrack Search

The chronological backtracking algorithm orders search using its **variable** and **value ordering** heuristics. The heuristics used can vary in sophistication: they can be general, or take into account problem specific characteristics; and they can fix orderings at the start of the search (static ordering), or update the ordering as the search progresses (dynamic ordering). We describe briefly those relevant to the thesis.

One of the most important sources of inspiration for search ordering strategies is the **fail-first principle (FFP)** described by Haralick and Elliot [43]. This is based on the observation that searching the most constrained parts of a problem first allows the early detection of inconsistencies in the partial assignment. The best means to implement this principle may depend on problem specifics. However, several general FFP variable ordering strategies have been created, and are now established enhancements of backtrack search: the most well-known of these is the **minimal domain-size first (MDF)** heuristic. This variable ordering strategy selects the variable with the least number of elements in its domain, and is employed later.

Another widely applied principle for search ordering has recently been named the **succeed-first principle (SFP)** by Smith [83]. This prefers the search choice that moves the search to the least constrained sub-problem, in order to maximize the chance of finding a solution. Typically, this principle manifests itself as a value ordering heuristic that is applied after the tightest part of the problem has been targeted by an FFP variable ordering heuristic. The implementation of the principle is dependent on problem specifics. In contrast to the FFP, no standard heuristics have yet emerged. The third case study will use an SFP heuristic.

Consistency Methods for Backtrack Search

Consistency methods (also referred to as *constraint propagation* or *local consistency* algorithms) support **backtrack** by avoiding partial assignments that are inconsistent. Consistency methods may be applied as a pre-processing step (statically) or at run-time (dynamically). The consistency methods that are referred to in the thesis are introduced below.

Consistency methods aid **backtrack** in two ways. First, they may pinpoint and delete values in the domains of unassigned variables that would lead to only inconsistent extensions of the current partial assignment. Second, they can detect inconsistency in the current partial assignment, triggering a backtrack. In many consistency methods, the inconsistency of the current partial assignment manifests itself as an empty domain.

The levels of consistency enforced by consistency methods are defined here using the abstract notions of 1-consistency and k-consistency, due to Freuder [33].

Definition 2.3.1 *1-consistency*
An FCSP (V, D, C) *is* ***1-consistent*** *if and only if, for each domain* d_i *of* $v_i \in V$, *and each value* $val \in d_i$, *the assignment of* $v_i \in V$ *to* val *does not violate any of the unary constraints in* C.

1-consistency is also known as *node consistency* (NC). It is almost always established in an FCSP before search commences since it is computationally cheap. Its dynamic variant is *forward checking* (FC) [43], the simpler of the two consistency techniques hybridized in Case Study I.

Definition 2.3.2 *k-consistency*
An FCSP (V, D, C) *is* ***k-consistent****, where* $k > 1$, *if and only if, for each assign-*

ment α of length $k-1$ that does not violate any constraint in C, and for each variable v_i unassigned by α, α can be extended to an assignment of length k, that additionally assigns v_i to a value from d_i, without violating any constraint in C.

Establishing k-consistency in an FCSP can be computationally expensive: Cooper's general k-consistency algorithm KS-1 has time complexity $O(\sum_{i=1}^{k}(\binom{n}{i} \cdot d^i))$, where n is the number of variables and d is the size of the largest domain [12]. Hence most consistency methods seek to achieve k-consistency for only small values of k. In fact, the general notion of k-consistency was originally based on Arc Consistency and Path Consistency (PC) — two low degrees of consistency for which practical algorithms had been developed.

Definition 2.3.3 *Arc Consistency (AC)*
*An FCSP (V, D, C) is **arc consistent** if and only if it is 2-consistent.*

Definition 2.3.4 *Path Consistency (PC)*
*A binary FCSP (V, D, C) is **path consistent** if and only if it is 3-consistent.*

Methods designed to achieve AC and PC in an FCSP established the field of constraint satisfaction within AI. Waltz [96] and Montanari [66] devised methods that resulted in arc and path consistent constraint graphs. These methods and concepts were compared and elaborated in [60], who also named the first three AC methods AC-1, AC-2 & AC-3, and the first two PC methods PC-1 and PC-2. Since then, much research has been conducted to improve the efficiency of local consistency methods, and of arc consistency in particular [65, 21, 92, 6, 18, 8, 80]. [65] is of special interest to us, since it describes the arc consistency procedure, AC-4, which is hybridized with FC in Case Study I. Case Studies II & III will use other arc consistency methods, including a class that establishes a "looser" level of arc consistency in problems with numeric constraints. Methods in this class obtain arc consistent

lower and upper bounds for numeric variables, and are collectively known as arc-B consistency methods [58].

Although the above methods were initially aimed at reducing the problem before search commenced, NC, AC, and increasingly PC are more likely to be applied dynamically during search. At runtime, the consistency methods can take into account the latest changes to the problem being searched. For example, in chronological backtracking, extending the current partial assignment by assigning a new variable yields changes in the constraints of the problem. All unary constraints on the assigned variable may be removed, while $n(>1)$-ary constraints on the variable are effectively reduced to $(n-1)$-ary constraints. Dynamic variants of node consistency (i.e. forward checking), arc consistency, and path consistency, may be applied to *maintain* those levels of consistency in the changed problem.

Learning Methods for Backtrack Search

We mention learning methods very briefly since they are not explored in the thesis. Learning methods record information between backtracks to avoid repeatedly searching the same sub-problems. Learning methods include *no-good recording* methods [16, 63, 81, 76], and *intelligent backtracking* strategies such as conventional or constraint-directed backjumping [37, 72] and dynamic backtracking [38].

2.4 Alternative Search Strategies

We now introduce several other search strategies. Some of these are used later to support backtrack search in the hybrids presented, while the remainder are referred to in discussions and comparisons.

2.4.1 Local Search

Unlike the chronological backtrack search algorithm, most local search strategies associate every search node with a *complete* assignment. The search commences with an initial assignment that is, in general, inconsistent. The task of the search is to change it with the aim of transforming it into a solution. Case Study II will describe a hybrid algorithm that integrates a local search method with chronological backtracking such that performance improvements are obtained.

A local search strategy is described by:

- an **initialization routine**, which is a cheap (usually polynomial time) algorithm that uses heuristics to produce an initial good quality assignment that can be used as the start search node;
- a **neighbourhood operator**, which generates the set of search nodes that can be visited from the current node;
- and a **neighbour ordering strategy**, which is used at every search node to select one of the neighbours generated by the neighbourhood operator.

One of the simplest examples of local search in CP is based on Pearl's general hill climbing algorithm [70]. This is *min-conflicts*, the heuristic repair algorithm of Minton et al. [64], which applies hill climbing to constraint violation repair. The min-conflicts search heuristic combines a neighbourhood operator and a neighbour ordering strategy. The neighbours of a search node are those that assign one of the "conflict variables" (those participating in a violated constraint) differently from the present assignment. The neighbourhood ordering strategy selects one of the conflict variables randomly. Each of the values in its domain is then evaluated, according to the total number of constraints that are violated when the variable is assigned to it. The value that violates the fewest constraints replaces the old assignment

for that variable, completing the move to a neighbouring search node. The search terminates after a pre-determined number of moves, or when the search is found to be stuck in a *local minimum*, where all the neighbours are worse quality than the present search node.

Most local search algorithms for constraint satisfaction are similar to heuristic repair but include additional mechanisms for avoiding local minima. Some use dynamic mechanisms that allow the algorithm to escape local minima — for example the connectionist algorithm GENET [15] and its descendant guided local search (GLS) [94]. Many are related to algorithms originally designed primarily for optimization. These have built-in mechanisms for breaking out of local minima, and include simulated annealing [54], tabu search [39, 40] and genetic algorithms [47]. Such optimization algorithms can be adapted for constraint satisfaction by incorporating a measure of solution consistency into their optimization functions, possibly in addition to any optimization requirement.

Notice that in general, a local search algorithm is neither (sat- nor opt-) sound nor (sat- nor opt-) complete:

- Normally, local search algorithms are *not sat-sound* and *not sat-complete* since minimizing constraint violations by optimization does not guarantee their elimination.[6] A (non-trivial) local search algorithm could potentially be sat-sound and -complete, but only if (1) the initialization procedure is sat-sound and sat-complete; and (2) the neighbourhood procedures guarantee not to subsequently violate any of the problem constraints. However, for complex problems, it is difficult to find simple initialization and neighbourhood procedures that can do so.

[6]Sat-soundness could be trivially guaranteed by not outputing any assignments that violate constraints, but, unless there is a high probability that the local search algorithm will find such assignments, this is not practical.

- A local search algorithm is generally *not opt-complete* and not *opt-sound*, since local search does not (ordinarily) perform exhaustive search. Exhaustive search is a pre-condition to achieving these properties in non-trivial algorithms. It is possible to integrate exhaustive search if, firstly, the visited search nodes are recorded and subsequently avoided, and, secondly, the neighbourhood procedures guarantee that the solutions always remain reachable. However this is difficult for two reasons. Firstly, ordinary local search is non-systematic, and this makes it space-inefficient to record all the visited search nodes ($O(d^n)$, where d is the size of the largest domain and n is the number of variables). Secondly, it is usually difficult to create neighbourhood operators that exclude visited search nodes, while guaranteeing the algorithm can still reach the solution nodes. One unconventional mechanism to restore opt-completeness and opt-soundness is to use systematic neighbourhood operators, which also offer the advantage space-efficient operation. This option will be explored in Case Study II.

To summarize, soundness and completeness are properties rarely found in local search algorithms for constraint satisfaction and optimization. Nevertheless local search algorithms are capable of solving many large and complex problems where a backtrack search algorithm would not terminate in an acceptable time. Unlike most backtrack search algorithms, the search can change the value assigned to a variable at any point in the search, which diminishes the chance that the algorithm focuses only on unpromising regions of the search space. Local search algorithms are especially suited to solving problems with high solution densities (proportions of solution space to search space), exchanging theoretical soundness and/or completeness for practicality and average-case efficiency.

2.4.2 Dual Assignment Search

Dual assignment is a term introduced here to describe existing algorithms that simultaneously work on two assignments. The first assignment is a partial, consistent assignment similar to that of backtrack search. The second assignment is sometimes designated a *probe*, and is often complete. The aim is to use the conflicts found in the probe to direct the consistent extension of the first assignment. Case Study II will deploy a simple dual assignment method based on Harvey and Ginsberg's *limited discrepancy search* [44]. Case Study III will demonstrate a hybridization form that is itself a dual assignment strategy.

Dual assignment algorithms are a relatively recent development in the CP literature. One of the first instances was Minton et al.'s *informed backtracking* [64]. This uses a heuristic procedure to generate an initial probe assignment. The probe assignment is then repaired using a chronological backtracking procedure, in conjunction with the min-conflicts heuristic. Yokoo also combined min-conflicts with a dual assignment strategy in *weak commitment* search [98]. Here chronological backtracking is replaced by a procedure that always backtracks to the root node. A no-good learning component is used to restore completeness, by recording previously visited search nodes. Another example is to be found in Verfaillie and Schiex's *local changes* algorithm, [93] which recursively operates on two partial assignments.

Dual assignment also has precursors in mathematical programming. *Mixed integer programming* (MIP) (Section 2.4.3) uses a solution to the "continuous relaxation" of a problem as a probe, and incrementally repairs the "integer" constraints.

All the above algorithms demonstrate that supplementing backtrack search procedure with a probe repair method can result in an improved algorithm. Walton supports this evidence by proving that the repair of (even) a random probe assignment improves backtrack search's average case performance [73].

2.4.3 Mathematical Programming Methods

Among the range of mathematical programming tools and methods we detail, two are directly relevant to the hybridization methods and the empirical studies presented later. These are *linear programming* (LP) and one of its extensions, *mixed integer programming* (MIP).

Linear Programming

An LP solver will be hybridized with backtrack search in Case Study III. We give here a definition of the LP problem that is addressed by LP solvers.

In the following, let a **linear term** on the set of variables $\{v_1, \ldots, v_n\}$ be an expression of the form

$$c_o + c_1 v_1 + c_2 v_2 + \cdots + c_n v_n, \qquad \text{where each } c_i \text{ is a constant}$$

Definition 2.4.1 *Linear Programming (LP) Problem*
A ***linear programming problem*** *is a CSOP* (V, D, C, g) *such that: every domain* d_i *of* $v_i \in V$ *is continuous; the constraint set* C *includes all constraints in the set* $\{v_i \geq 0 | v_i \in V\}$*; every other constraint in* C *is in the form* $X \leq Y$ *or* $X = Y$*, where* X *and* Y *are linear terms on* V*; and* g *can be expressed as a linear term on* V*.*

In general, the LP problem allows unbounded domains i.e. ones having an infinite upper bound, but in the thesis the LP domains are always bounded.

LP problems are a tractable class of CSP for which polynomial algorithms have been found [53, 52]. However, (worst-case exponential) Simplex-based algorithms are more usually applied, and in particular the Two-Phase Primal or Dual Simplex

algorithms (see [13]). This is because for most problems they still have better average case performance than their polynomial counterparts [9].

Mixed Integer Programming

Case Studies II & III use MIP algorithms to gauge the performance of several of the hybrid algorithms described in the thesis.

The MIP problem extends the LP problem to allow the representation of the integrality constraint, which forces a variable to take an integer value. This extension is used to model problems with discrete variables and/or problems with disjunctions.

Definition 2.4.2 *Mixed Integer Programming (MIP) Problem*
A ***mixed integer programming problem*** *is a CSOP* (V, D, C, g) *such that: every domain* d_i *of* $v_i \in V$ *is continuous; the constraint set* C *includes all constraints in the set* $\{v \geq 0 | v \in V\}$*; every other constraint in* C *is in the form* $X \leq Y$ *or* $X = Y$ *or* integer(v)*, where* X *and* Y *are linear terms on* V*,* $v \in V$*, and* integer(v) *is a constraint requiring* v *to take only integer values; and, finally,* g *can be expressed as a linear term on* V*.*

Note that MIP can capture any problem expressed as a minimization (or maximization) of a linear optimization function over a disjunction of LP problems. MIP problems are still a restriction of the CSOP because they cannot express non-linearity (e.g. quadratic terms), either in the optimization function or in the constraints. Nevertheless, it is theoretically possible to transform problems involving only disjunctive constraints and linear optimization functions into MIP problems.

The theory and practice of MIP is described well in a number of introductory texts (see in particular [82, 36]). We note here only that the core of all major MIP methods

is an LP solver. It follows that a large proportion of the constraints must be linear before effective solution is possible by MIP methods.

2.5 Summary

This chapter related the definitions of the constraint satisfaction and optimization problems. The backtrack search strategy was described in some detail, together with the principal methods that will be used to support it in the hybrid algorithms of the thesis. Other techniques that will be referred to later were introduced.

Chapter 3

The Hybridization Context

Before presenting the three case studies, this chapter reviews aspects of the hybridization context referred to in Chapter 1.

3.1 The Hybridization Conditions

Chapter 1 described four conditions that defined the character of the hybrid algorithms investigated in this thesis. We discuss below the reasons for adhering to these conditions.

Condition 1: Backtrack search control

The hybrid algorithms consist of a backtrack search procedure controlling and coordinating search, with a set of hybridized "support methods" assisting it.

Basing the hybrid algorithms on backtrack search is a good choice for two principal reasons:

- Backtrack search is both generic and flexible, requiring little customization before application to ordinary constraint satisfaction problems. Traditionally, refinements are made through the incremental revision of the search ordering heuristics and the integration of local consistency methods. The versatility of backtrack search has made it the CP algorithm of choice: the majority of CP languages, including CHIP [22], ILOG Solver [50], and ECL^{i}PS^e [49], use backtrack search as their default search engine. For similar reasons, constructing hybrid algorithms around a backtrack search core can facilitate the process of algorithm development and result in flexible and extensible hybrids.

- Backtrack search is sound and complete for constraint satisfaction (and may be extended to be so for optimization), and yet it remains space-efficient (polynomial in the worst-case). This is due to the systematic manner in which search decisions are made. By comparison, complete, non-systematic search algorithms have exponential worst-case space complexities [75].

Unfortunately, the major drawback of backtrack search is its time inefficiency (worst-case exponential). In the thesis, the hybridization of "support methods" is the means by which time efficiency is improved.

Condition 2: Localized Hybridization

The support methods are localized: they are brought in to assist backtrack search on specific sub-problems.

Localized hybridization is the application of specialized support methods to particular sub-problems. If necessary, sub-problems may overlap, and/or change as the search progresses.

Localized hybridization is sometimes vital to good algorithm performance, as ex-

plained in the discussion on LSCOs in Chapter 1. It depends on the successful implementation of three steps: the decomposition of the problem into sub-problems with different characteristics; the mapping of appropriate methods to the sub-problems; and the productive integration of their outputs. The three steps can be difficult to implement when the sub-problems are inter-dependent, and/or when the sub-problems are made to change during search.

We restrict our investigations to hybrids that use some form of localized hybridization to improve efficiency, with the objective of investigating these steps for different problem and algorithm combinations.

Condition 3: Dynamic Hybridization

The support methods are dynamically hybridized: they co-operate with backtrack search during its execution.

In other words, the support methods are not simple pre-processing or post-processing steps that execute before or after the backtrack search. The reason for preferring dynamic hybridization is that it enables a tighter — and therefore potentially more efficient — integration of the support methods with backtrack search. Support methods can supply up-to-date information to the backtrack search at different search nodes, creating opportunities for synergistic behaviour, and ultimately enabling more intelligent search.

Condition 4: Soundness and Completeness

The basic hybrid algorithm is sound and complete — with one small caveat.

Let us consider first soundness and completeness for a *constraint satisfaction* algorithm. Normally, soundness is an essential property, since an unsound algorithm

may not generate true solutions. In addition, algorithm completeness is desirable, since proofs of problem infeasibility are useful.

This condition stipulates that when hybridizing backtrack search with the support methods, the basic hybrid algorithm is sound and complete, *with one small caveat.* The caveat is because the requirement for completeness may limit the algorithm's potential when problem scale is large. In such cases, an exhaustive search for solutions is not a realistic option, and the completeness of the hybrid algorithm should be sacrificed to improve search efficiency.

However, by starting with a complete hybrid algorithm, completeness may later be traded-off for efficiency in a calculated and controlled manner, via the curtailment or abandonment of search in especially selected regions. Case Study II will demonstrate one way in which incompleteness can be introduced. Adhering to this condition allows a degree of control over the type and scale of incompleteness that would not be possible if the basic hybrid algorithm were inherently incomplete.

This approach can be carried forward for *constraint satisfaction and optimization* algorithms. It was noted in Section 2.3.3 that backtrack search may be simply extended to be sound and complete for optimization. However, if problem scale is large, it is possible to sacrifice the requirements of opt-soundness and opt-completeness, for example by outputing sub-optimal solutions, or by imposing time-outs. However, by adhering to this condition, we ensure that (as for backtrack search) our hybrid algorithm could also be used for sound and complete optimization.

3.2 Support Methods for Backtrack Search

In the hybridization context established by the four conditions, the aim of hybridization is to bring about reductions in the size of the search performed by the backtrack

algorithm, by appropriately applying "support methods" to parts of the problem. We loosely define a **support method** to be any technique that may be integrated with the basic backtrack search algorithm to improve its performance.

In Chapter 2 the traditional approaches used to enhance backtrack search were classified as 1) search ordering heuristics, 2) consistency methods, and 3) learning & intelligent backtracking methods. We extend these approaches with a fourth enhancement category that captures some types of hybridization within our hybridization context. This fourth category contains methods that independently search sub-problems and report results that may be used to aid the backtrack search. We name the methods in this category **4) forward search** methods.

Forward search methods solve sub-problems with the objective of guiding the backtrack search and reducing its size. In the hybridization context, forward search methods suggest assignments for unassigned variables, given the backtrack decisions made so far. For example, a linear programming solver might be applied to obtain a suggested assignment that satisfies some of the problem's linear constraints. The suggested assignment may thus be only partial and/or partially consistent. Nevertheless, such suggested assignments can focus the backtrack search and greatly limit its extent. Case Studies II & III will demonstrate two basic ways to integrate forward search methods to obtain efficiency improvements. Several examples will also be noted from the OR and CP literature that (implicitly) utilize these two basic integration techniques.

3.3 Hybridization Issues

The nature of the hybridization conditions means that several key issues must be tackled when constructing hybrids. The most important of these will be the subject of discussion in the thesis, and are introduced here.

3.3.1 Selection & Scoping

Selection refers to the process that decides the set of support methods applied at any point in the search. *Scoping* is the process that chooses, for each selected method, the sub-problem it is applied to: this sub-problem is designated its *scope*.

When constructing selection and scoping strategies for a hybrid algorithm a number of issues must be considered:

Compatibility. Clearly the scope chosen for a given support method must respect the class of problems it can handle. For example, a linear programming solver is incompatible with a scope that includes disjunctive constraints.

Efficiency. Even when a method can address a particular scope, it may not be appropriate from an efficiency point of view. The quality of the selection and scoping strategies is measured by the efficiency of the hybrid algorithm as a whole. This is very dependent on the levels of synergy that can be achieved among the various support methods, and the added value they collectively bring to the backtrack search.

Complexity. Constructing a hybrid algorithm can be a complex affair. This is especially so when the chosen scopes of methods overlap, potentially complicating integration, or when the selected methods and their scopes are made to change dynamically during search.

3.3.2 Synchronization & Integration

Synchronization refers to the co-ordination of support methods, such that, at any point in the search, they are always solving parts of the same problem. In the hybridization context this issue is simply addressed: whenever a new search decision is made by the backtrack search it is communicated to all the support methods. Similarly, whenever a backtrack is executed by the backtrack search it is reported to all the support methods.

Once synchronized, the support methods are activated, generating various search information. *Integration* is the process by which this information is incorporated back into the backtrack search. The integration issue is *how to use search information to improve algorithm efficiency, without compromising algorithm soundness and completeness.*

Clearly, the type of search information generated by a support method dictates the way in which it must be integrated into the backtrack search. For example, when an arc consistency method generates domain reductions at a search node, these domain reductions can be applied without any loss of algorithm soundness or completeness. Contrast this with a suggested assignment generated by, for example, a linear programming forward search method. It is unlikely that the suggested assignment can be applied without endangering both soundness and completeness.

Recall that the basic backtrack search algorithm utilizes available search information in three different stages, named earlier the **consistency**, **branching** and **decision** stages (Section 2.3). The hybrid algorithms of the thesis integrate the different types of search information within these three stages, such that algorithm soundness and completeness are preserved.

3.4 Remainder of the Thesis

The remainder of the thesis will explore synchronization & integration, and selection & scoping, via the construction of hybrid algorithms addressing specific problem classes. Each of the algorithms introduces hybridization at different backtrack search stages.

In Case Study I, the hybridization takes place in the *consistency stage* of backtrack search. The case study explores the hybridization of local consistency support methods — viz. arc consistency and forward checking — with backtrack search. The problems addressed are binary constraint satisfaction benchmarks. Dynamic scoping is applied to share the responsibility of constraint checking among the support methods such that efficiency gains are obtained.

Case Study II applies complete, repair-based algorithms as forward search methods. These methods are statically scoped, and suggest consistent but only partial assignments. The methods take over — for particular sub-problems — the entire search process: they replace the backtrack search's *consistency, branching and decision stages* on those sub-problems. By breaking-up the problem addressed by backtrack search, the forward search methods can obtain big efficiency improvements, without compromising completeness in the basic algorithm.

The largest case study, Case Study III, addresses a class of dynamic scheduling problems. Suggested assignments are generated by a forward search method, a Simplex-based linear solver. The solver is dynamically scoped to suggest partially consistent assignments that are the key drivers of the search. Since they are only partially consistent, these assignments are integrated differently: these assignments direct and limit the *branching and decision stages* of backtrack search. The resulting tight integration is found to obtain significant performance improvements, while maintaining soundness and completeness.

CASE STUDY I

Chapter 4

Adaptive Propagation

The material in the chapter was first published in [30].

4.1 Introduction

4.1.1 Overview

Algorithms for maintaining consistency, also known as *constraint propagation* methods, vary in the levels of consistency they achieve. This case study investigates **adaptive propagation** – the process of varying the strength of constraint propagation (the level of consistency achieved) to reflect the dynamics of search. The problem addressed is **binary constraint satisfaction**.

Here, the focus is on hybridizing two of the simplest propagation methods, arc consistency (AC) and forward checking (FC) (Section 2.3.4). AC-based algorithms apply a stronger form of propagation than FC-based algorithms; they invest greater computational effort to detect inconsistent values earlier. Thus the relative payoff

of maintaining AC during search as against FC varies for different constraints and for different search states.

A scheme is presented for *adaptive arc propagation* (AAP) that allows the *dynamic scoping* of these two consistency methods. Under the scheme, meta-level reasoning and heuristics can be used to distribute dynamically propagation effort between the two. An instance of AAP, *anti-functional reduction* (AFR), is described in detail. It is used to dynamically scope a particular AC method, AC-4. AFR achieves precisely the same propagation as the pure AC method, while improving its performance. The strategy is to reduce gradually the scope of AC propagation during backtrack search, to exclude those constraint arcs that may be subsequently handled as effectively by FC. Experimental results on binary constraint satisfaction benchmarks confirm that performance improvements are achieved by AFR. More generally, these results demonstrate that the dynamic scoping of local consistency checking methods can improve efficiency.

4.1.2 Background

The strategy of using knowledge of the nature of the constraints and variables in a CSP (meta-level knowledge or meta-knowledge) to reduce the computation required to achieve full arc consistency in a constraint graph has long been advocated [92, 34, 7]. However, the assumption has been that problem characteristics are known at the start of search.

As explained in Section 2.3, chronological backtrack search enhanced with consistency methods such as forward checking (FC) or arc consistency (AC) involves the repeated application of two alternating steps: labelling (setting an unassigned variable to a value from its domain) and constraint propagation (the deletion of inconsistent values from the domains of the remaining unassigned variables). Even a single

application of this two-step label-propagate process results in a transformation of the original problem to a sub-problem with one or more reduced domains. Thus initial meta-knowledge (about structure, constraint density & tightness, etc.) can be rendered obsolete, since sub-problems will have characteristics that are different from those of the original problem.

The importance of using up-to-date (i.e. dynamically changing) meta-knowledge to improve the performance of search is well illustrated by comparing the effectiveness of static and dynamic search ordering methods. For example, the minimal domain-size first (MDF) variable ordering heuristic ([43], Section 2.3.4) is much more effective than its equivalent static ordering heuristic. This case study investigates whether dynamic meta-knowledge can also be exploited in the context of consistency methods.

Some research has recently been conducted into dynamically adapting algorithms to the problem undergoing solution. Performance benefits have been achieved by substituting one algorithm for another at run-time, using indicators of algorithm performance as a guide [10]. Usually the algorithm substitution simply corresponds to a change in the methods selected to support backtrack search: thus one consistency method might be substituted for another during search.

However, little research has studied the dynamic alteration of the scopes of several alternative consistency methods deployed simultaneously within a hybrid framework (adaptive propagation).[1] The goal of the case study is to apply dynamic meta-knowledge to do so.

Specifically, the basis of hybridization is a scheme named adaptive arc propagation

[1] Although Dechter & Pearl's *adaptive consistency* procedure ([20]) might appear to be relevant to our notion of algorithm adaptation, it is in fact not closely related. This is because it does not dynamically hybridize several distinct propagation methods: *adaptive consistency*, given a pre-determined static variable ordering, is a single consistency method that establishes a level of directional consistency that avoids backtracks in the subsequent "search".

(AAP), designed to allow the dynamic scoping of AC and FC techniques on binary constraint satisfaction problems. AAP decides an initial status for each binary constraint arc that determines how changes to the "source" variable's domain will be propagated onto the "destination" variable. For AC, propagation occurs on any change to the domain of the source variable, while for FC, propagation occurs only on its instantiation to a single value. As labelling and propagation take place, the algorithm reviews relevant meta-knowledge and uses it to change the status of these constraint arcs, reducing or expanding the scopes of AC and FC.

Earlier algorithms have hybridized FC and AC, but these hybridizations were *static* in the sense that they applied FC and AC in pre-determined phases, and did not update their configuration according to the dynamics of the search. This form of hybridization seems to be less effective than pure AC for hard problems [78].

The instance of AAP described here, anti-functional reduction (AFR), dynamically scopes a particular AC method, AC-4 [65], hybridizing it with FC. The resulting hybrid algorithm can be viewed as an improved variant of AC-4. Empirical results for AFR demonstrate the viability of adaptive propagation, and, more generally, illustrate the utility of dynamic scoping within our hybridization framework.

4.1.3 Case Study Outline

The remainder of the case study is structured in three descriptive sections followed by experimental results and a conclusion. Section 4.2 details the difference between AC and FC propagation to suggest how the appropriate method may be selected for particular constraint arcs. Adaptive arc propagation (AAP) is described in Section 4.3. Section 4.4 then gives an instance of AAP, anti-functional reduction (AFR). The results for AFR (Section 4.5) confirm performance improvements over algorithms performing pure AC-4 and FC propagation. Section 4.6 concludes the

case study.

4.2 Arc Consistency and Forward Checking

Arc consistency is generally more powerful than forward checking in that it provides earlier detection of inconsistent values. The exact conditions under which arc consistency is superior to forward checking in binary CSPs are described in this section.

4.2.1 Basic Notation

The notation given below is used in the following sections. Let X and Y be two CSP variables.

- d_X the domain of a CSP variable X.
- $|X|$ is the size (cardinality) of d_X.
- The values $x_1..x_{|X|}$ represent the elements in d_X.
- $r(X, R)$, where R is a subset of d_X, denotes an algorithmic operation that transforms the domain of X in the CSP from d_X to a reduced domain d'_X, such that $d'_X = d_X \backslash R$. Such transformations are known here as *reductions*, and are applied during constraint propagation.
- A binary constraint $constr_{\{X,Y\}}$ on X and Y is represented by the two directed arcs arc_{XY} and arc_{YX} corresponding to the two directions in which the constraint can propagate.

The set of all constraint arcs will be referred to as $Arcs$.

4.2.2 FC- and AC-monitoring

The behaviour of AC and FC on a directed arc arc_{XY} will be compared by counting the number of different possible reductions on X for which AC would yield a reduction on Y, but FC would not. The terms used in the discussion are introduced below:

- A reduction $r(X, I)$ is said to be an *instantiation* of X when $|I| = |X| - 1$.
- An arc_{XY} is said to *propagate* a reduction $r(X, R)$ of X when the domain of Y is reduced to exclude those values inconsistent with X's new domain and $constr_{\{X,Y\}}$.
- An arc_{XY} is said to be *FC-monitored* in a propagation sequence if it is used only to propagate those reductions of X that are instantiations.
- An arc_{XY} is said to be *AC-monitored* if it propagates any reduction of X.

4.2.3 When is AC More Effective than FC?

Constraints are relations and may be represented in terms of a relation matrix. Matrices are used to analyse FC and AC propagation.

Example. Let the constraint $constr_{\{X,Y\}}$ (corresponding to $X \leq Y$ when X and Y have the domain $\{1, 2, 3\}$) be described by the following tuple representation:

$$\{(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)\}$$

The equivalent relation matrix is given in Table 4.1.

Table 4.1: An example relation matrix (describing the constraint $X \leq Y$)

$X \leq Y$	X value 1	2	3	ones	zeros	Minimum monitoring of arc_{XY} required to ensure $y_i \in d_Y$ is deleted as soon as its support in d_X is lost
Y value 1	1	0	0	1	2	AC
2	1	1	0	2	1	FC
3	1	1	1	3	0	None

In the table, 0-1 Boolean values indicate whether a pair of values hold in the constraint relation. In addition the example matrix is annotated with the quantities of **ones** in a given row. These quantities are referred to in the arc consistency literature as *support* for a value and are denoted here by $S_X(y_i)$ [65]. In our comparison of arc consistency and forward checking the important quantity is the number of **zeros**; this quantity is named the *support-complement*. The support-complement $\overline{S}_X(y_i)$ is related to $S_X(y_i)$ and $|X|$:

$$\overline{S}_X(y_i) = |X| - S_X(y_i) \tag{4.1}$$

Consider the result of arc_{XY} propagating reductions from X to Y. A value is deleted by AC-monitoring from the domain of Y when it loses all its support in the domain of X. So, in Table 4.1 the value 1 is deleted from the domain of Y if one of the following sets of values were to be removed from the domain of X: $\{1\}, \{1,2\}, \{1,3\}$, or $\{1,2,3\}$. Notice that the removal of values $\{1,2,3\}$ from the domain of X would yield an empty domain, and as such would trigger immediate backtracking: it never needs to be propagated by AC-monitoring, nor by FC-monitoring. Notice also that removing the values $\{1,2\}$ or $\{1,3\}$ from the domain of X would instantiate X. Therefore, for these two reduction sets, FC-monitoring would also detect the fact that value 1 can be deleted from the domain of Y. Only the deletion of $\{1\}$ from the domain of X would cause AC-monitoring to delete value 1 from the domain of Y, when FC-monitoring would not.

We now generalise this analysis. AC-monitoring is superior to FC-monitoring when a reduction of X that is not an instantiation results in a reduction of Y. Those values of Y that have two or more zeros in their row are the only candidates for deletion in such a situation (e.g. value 1 in the table). This is because values that have no zeros (e.g. value 3) are always consistent with the constraint arc, while those with a single zero cannot be deleted until X is instantiated (e.g. value 2). Thus only when $\overline{S}_X(y_i) > 1$ can AC-monitoring of arc_{XY} result in the removal of y_i before FC-monitoring.

The final column of Table 4.1 gives the weakest level of propagation for $arc_{\{X,Y\}}$ that ensures y_i is removed from d_Y as soon as it loses all its support in d_X (thus obtaining the same level of propagation achieved by AC). The table shows some values which need weaker levels than AC (viz. values 2 and 3 in the table).

An arc arc_{XY} is said to be *anti-functional* if and only if $\forall y_i : \overline{S}_X(y_i) \leq 1$. The class of constraints $constr_{\{X,Y\}}$ that have anti-functional arc_{XY} and arc_{YX} are known as *anti-functional constraints* [92]. The disequality constraint $\neq$ is an example of an anti-functional constraint.

4.2.4 AC-superiority

In Table 4.1, consider again value 1 in Y's domain. The associated support complement $\overline{S}_X(1) = 2$, while number of reductions of X that would cause its deletion is 2^2 ($\{1\}, \{1,2\}, \{1,3\}$ and $\{1,2,3\}$ — note that 2^2 is actually the cardinality of the powerset of {2,3}). Eliminating $\{1,2\}$ & $\{1,3\}$ (because they are both instantiations), and $\{1,2,3\}$ (because it never occurs), yields the number of reductions for which AC-monitoring is superior to FC-monitoring: $2^2 - 2 - 1 = 1$.

Let us assume a situation where all possible reductions are equally probable in a propagation sequence. If all reductions were equally probable, we could deduce a

figure for the probability of AC-monitoring deleting a value earlier. For each arc arc_{XY}, the probability of AC-monitoring deleting some value $y_i \in d_Y$, when FC-monitoring would not, can be calculated. We name this probability *AC-superiority*, and define it below.

Let k denote the value for some support complement $\overline{S}_X(y_i)$. When $k > 1$, the number of possible reduction sets that are not instantiations of X, but result in the deletion of y_i in AC-monitoring but not in FC-monitoring, is given by:

$$n_1 = 2^k - k - 1 \tag{4.2}$$

On the other hand, the number of potential reductions of $X, |X| > 2$ is equal to the cardinality of its powerset: $2^{|X|}$. Of these reductions, $|X|$ are instantiations, 1 is 'empty' (it is the empty set), and 1 is the set d_X. Hence the number of 'non-empty' reductions that do not lead to an instantiation of X, nor to its having an empty domain, is obtained as follows:

$$n_2 = 2^{|X|} - |X| - 2 \tag{4.3}$$

Thus AC-superiority depends only on $|X|$ and the support-complement of y_i :

$$AC\text{-}superiority(y_i) = \frac{n_1}{n_2} \tag{4.4}$$

Figure 4.1 shows graphically how AC-superiority can vary with support-complement

values.

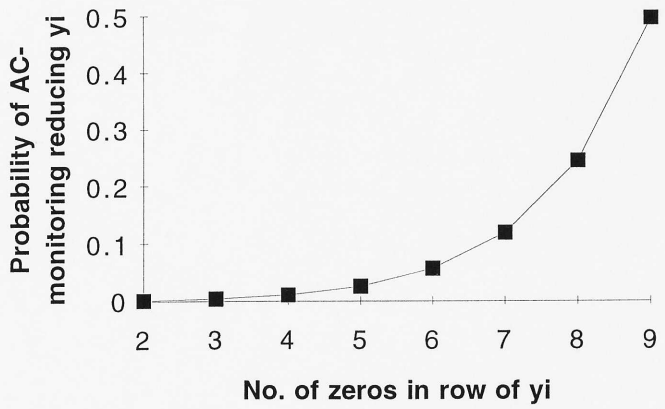


Figure 4.1: AC-superiority vs. support-complement ($|X| = 10$)

The criterion used above to measure arc consistency superiority vis-à-vis forward checking was the number of propagated reductions. The analysis might be used to identify arcs with a very low probability of achieving superior propagation with AC-monitoring. AC-superiority figures could be obtained at run-time for particular values of y_i and $|X|$, and a decision procedure could then use this meta-knowledge as a heuristic, switch arcs propagating onto domains with low AC-superiority values from AC-monitoring to FC-monitoring.

Anti-functional reduction (AFR), the technique described later in Section 4.4 does not apply a sophisticated probabilistic decision procedure; only those arcs that are *guaranteed* to be handled as effectively by FC-monitoring are switched from AC-monitoring to FC-monitoring.

4.3 Adaptive Arc Propagation

Adaptive arc propagation (AAP) is an adaptive propagation scheme that makes use of arc tags to scope AC and FC propagation methods in a constraint graph. It is based on a generic scheme for arc consistency that is known as AC-5 [92]. Note

that, unlike AC-4, the AC-5 scheme is not a fully fledged arc consistency algorithm: the scheme simply defines how reductions are queued, but allows the definition of specialized propagation procedures for different constraints. The AAP scheme extends the AC-5 scheme to enable the dynamic scoping of AC and FC propagation methods.

Specifically, AAP adds initialization and maintainance procedures for the arc tags. The arc tags are updated dynamically to take into account dynamically changing AC and FC scopes. Anti-functional reduction will be used to demonstrate scoping in AAP (Section 4.4), but other instances could also be applied.

In the following, all procedures and functions that need to be defined by AAP instances are subscripted by inst. These include the two scoping procedures, $\mathsf{InitList}_{\mathsf{inst}}$ and $\mathsf{ReviseList}_{\mathsf{inst}}$, which respectively create and modify the scoping lists which contain the arc tags used in the scoping process. One list of tags is created for each arc $\{\mathrm{l}(X,Y)|arc_{XY} \in Arcs\}$.

$\mathsf{InitList}_{\mathsf{inst}}$ initializes each $\mathrm{l}(X,Y)$; it might assign it a default list or take into account initial knowledge about the arc in question. During backtrack labelling and propagation, whenever an arc is selected to propagate a variable reduction, the arc's list is first revised using the procedure $\mathsf{ReviseList}_{\mathsf{inst}}$ to take into account up-to-date meta-knowledge. Thus each arc propagation step involves also a list revision step, giving dynamic scoping mechanisms a fine grained control over propagation. Once this is completed, the first *active* propagation method in the list is applied – a method is active when its triggering conditions are satisfied.

4.3.1 AAP Initialization

Figure 4.2 gives the AAP initialization scheme. As described, the scheme is an extension of AC-5's. The procedures $\mathsf{InitMethod}_{\mathsf{inst}}$, Remove and Enqueue are sim-

ilar to their equivalents in the AC-5 scheme. $\mathsf{InitMethod}_{\mathsf{inst}}$ calls the appropriate initialization routines of the various propagation methods in the system, yielding a reduction, Δ_{out}. Remove implements the reduction by deleting the relevant values from the domain of the destination variable and Enqueue adds the new reduction to the queue of reductions that must be propagated. In addition to initializing the propagation method data structures, it is now made to apply the dynamic scoping procedure $\mathsf{InitList}_{\mathsf{inst}}$ for each arc. This creates the arc's scoping list, $l(X, Y)$.[2]

---

begin AAP-Initialization
    Q = {};
    for each $arc_{XY} \in Arcs$ **do**
        begin
            $\mathsf{InitMethod}_{\mathsf{inst}}\ (X, Y, \Delta_{\text{out}})$;
            Remove$(\Delta_{\text{out}}, d_Y)$;
            Enqueue$(Y, \Delta_{\text{out}}, Q)$;
            $l(X, Y) = \mathsf{InitList}_{\mathsf{inst}}\ (X, Y)$
        end;
end AAP-Initialization

Figure 4.2: AAP-Initialization

4.3.2 AAP Maintenance

AAP maintenance allows the revision of the lists created by $\mathsf{InitList}_{\mathsf{inst}}$ to reflect the search dynamics (Fig. 4.3). Dequeue obtains an arc and a domain reduction that requires propagation. One difference to the AC-5 scheme is that multiple reductions on a single arc are merged into a single reduction by Dequeue.[3]

The arc tag update procedure $\mathsf{ReviseList}_{\mathsf{inst}}$ is called to revise the scoping list as-

[2]Note that changes to the AC-5 initialization and maintenance schemes have been underlined in Figs. 4.2 & 4.3.

[3]This is a source of efficiency since reductions on a single arc are handled best together, minimizing queue operations and propagation steps.

```
begin AAP-Maintenance
      while not EmptyQueue(Q) do
            begin
                  Dequeue(Q, X, Y, Δ_in);
                  I(X, Y) = ReviseList_inst(X, Y);
                  ApplyActiveMethod(I(X, Y), X, Y, Δ_in, Δ_out);
                  Remove(Δ_out, d_Y);
                  Enqueue(Y, Δ_out, Q)
            end
end AAP-Maintenance
```

Figure 4.3: AAP-Maintenance

sociated with the arc. ApplyActiveMethod (Fig. 4.4) then traverses the updated list applying the first method for which the Boolean function $\text{Active}_{\text{inst}}$ returns true. This enforces an ordering of the applicable methods implicit in the list sequence, allowing them to be ordered according to their relative costs and benefits (see Section 4.4). The method chosen returns a possibly empty reduction Δ_{out} on the destination variable Y. The domain of Y is updated before any reductions are added to the queue as usual by Enqueue.

```
begin ApplyActiveMethod(l, X, Y, Δ_in, Δ_out))
      if empty(l)                          % no methods apply
        Δ_out = {}
      else
          method = head(l);
          if Active_inst (method, X, Y)
            then apply method(X, Y, Δ_in, Δ_out)
            else ApplyActiveMethod(tail(l),X, Y, Δ_in, Δ_out)
end ApplyActiveMethod
```

Figure 4.4: AAP propagation procedure

4.4 Anti-Functional Reduction

Anti-functional reduction (AFR) is an AAP instance that aims to improve the efficiency of the AC propagation method AC-4 [65], by hybridizing it with FC.

AC-4 conducts search by first performing all possible constraint checks to create a set of support counters (intended to keep an up-to-date record of the support values of Section 4.2.3). Once this is achieved, any values in a variable domain that have support counter values of zero can be removed. In the case of maintained AC-4, all subsequent activity consists of decrementing the support counters to reflect the changing problem, and removing values whenever their support counters drop to zero.

AC-4 is regarded as inefficient when used for *establishing* arc consistency in a constraint graph, because it always performs all possible constraint checks [95]. However, when it is used for *maintaining* arc consistency (MAC-4 [78]) in the constraint graph no constraint check needs to be repeated during search; all explicit constraint checks occur once and once only in the preprocessing phase. This is in contrast to other maintained AC methods which may need to repeat the same checks in different search branches.

Nevertheless, AC-4 must still perform support counter decrements during search, which are a source of inefficiency. The inefficiency stems from the memory management mechanisms of the backtrack search process. Every change that occurs to data structures (such support counters and variable domains) in a propagation sequence must be recorded to enable restoration of state on backtracking. The act of recording and restoring state during search consumes time and space, and so unnecessary propagation steps reduce the efficiency of the search algorithm.

When comparing AC and FC propagation over a single constraint arc, FC has far fewer record-able (restorable) operations than AC. There is no need as with AC to

record, potentially in every propagation sequence, multiple support counter decrements; only a single domain reduction when the source variable gets instantiated. Section 4.2 examined AC and FC propagation and described how it is possible to estimate the probability of AC-monitoring achieving hits (reductions of the destination variable) that would be missed by FC-monitoring. However, as mentioned earlier, AFR takes the approach that any possibility of a domain reduction is worth pursuing through the application of AC-monitoring. AFR identifies only anti-functional arcs that guarantee no loss in propagation when their monitoring status is switched from AC-monitoring to FC-monitoring, thus minimizing the amount of backtrack recording and restoration. The commonality between the data structures of AC-4, and those required to detect anti-functionality is key to the efficient detection of anti-functional arcs in AFR; the support-complement is obtained in terms of the support and the domain size of the source variable by Eqn.(4.1).

As well as dynamically detecting anti-functional arcs and switching them to FC-monitoring, AFR always activates FC instead of AC for reductions that are instantiations. These two enhancements reduce the number of restorable propagation operations in the average case, albeit with a small meta-knowledge maintenance overhead. The experimental results of Section 4.5 show that using FC to support AC in this way improves performance, especially for hard problems involving many backtracks.

4.4.1 AFR Initialization

As mentioned above, it is preferable to give priority to FC over AC when propagating reductions that are instantiations. A sensible initial ordering of applicable methods is thus {fc,ac} because it forces the use of forward checking in preference to arc consistency on instantiation (i.e. whenever $\mathsf{Active}_{\mathsf{afr}}(\mathsf{fc}, X, Y)$ returns true). Hence the function $\mathsf{InitList}_{\mathsf{afr}}$ defined in Fig. 4.5 assigns {fc,ac} as the initial scoping list

for all arcs that are not anti-functional.

```
begin InitList_afr(X, Y)
        d_Y^AC = {y_i : y_i ∈ d_Y ∧ S_X(y_i) < |X| − 1}
        if d_Y^AC = φ
          return {fc}
        else
          return {fc,ac}
end
```

Figure 4.5: AFR method list initialization

4.4.2 AFR Maintenance and Propagation

The task of the revision procedure of AFR is to identify when arc tags may be switched from mixed FC- AC-monitoring ({fc,ac}) to FC-monitoring ({fc}). The scoping list is revised by $\mathsf{ReviseList}_{\mathsf{afr}}$ according to the contents of a shadow domain d_Y^{AC} (Fig. 4.6). d_Y^{AC} contains all those values that are not handled as effectively by FC-monitoring of X. Note that d_Y^{AC} is monotonically reduced; once a value has been removed from d_Y^{AC} there is never a cause to reintroduce it to d_Y^{AC} in remaining sub-problems. When d_Y^{AC} is empty the switch can occur. It should be noted that any value that is not a member of d_Y^{AC} need never have its support counter $S_X(y_i)$ decremented by the AC-4 propagation procedure. The AC-4 procedure is made to reduce only the support counters of members of d_Y^{AC}, resulting in a further net reduction in the number of restorable propagation operations.[4] However, AC levels of propagation are guaranteed because the FC procedure takes control on instantiation, determining the status of those values not in d_Y^{AC}.

[4]Interestingly, this demonstrates that propagation can be scoped dynamically even within a single constraint.

```
begin ReviseList_afr(X, Y)
      d_Y^AC = {y_i : y_i ∈ (d_Y^AC ∩ d_Y) ∧ S_X(y_i) < |X| − 1};
      if d_Y^AC = φ
        return {fc}
      else return {fc,ac}
end
```

Figure 4.6: AFR scoping list maintenance

4.5 Empirical Results

4.5.1 Experimental Setup

Fifty thousand experiments were used to compare FC, AC, and AFR. A fixed variable ordering was employed to remove the influence of search ordering on algorithm performance. The experiments were conducted in groups of trials. The following parameters were varied over groups: no. of variables (10, 20), domain size (5, 10), and P_{density} (0.2, 0.4, 0.6, 0.8, 1.0). In addition, two tightness parameters were used $P_{\text{applicability}}$ (0.1,0.3,0.5,0.7,0.9) and $P_{\text{constriction}}$ (0.1, 0.3, 0.5, 0.7, 0.9).[5] The three probabilities are described below:

P_{density} Problem variable pairs (X, Y) have a constraint defined on them with probability P_{density}.

$P_{\text{applicability}}$ This parameter is used to control the proportion of values in variable domains to which the constraints apply. For each constraint $constr_{\{X,Y\}}$, two *applicability sets* (App_X and App_Y) are created. These contain the values in the domains of X and Y that the constraint applies to. A value is *not* a member of its variable's applicability set with probability $P_{\text{applicability}}$.

[5] Two tightness parameters were used instead of the single, conventional tightness parameter to increase control over the structure of constraints, as explained later.

Low values for $P_{\text{applicability}}$ imply tight constraints.

$P_{\text{constriction}}$ This controls tightness of a constraint $constr_{\{X,Y\}}$ given that it applies only to those values in App_X and App_Y. Two *tuple sets*, Tup_X and Tup_Y, are created. The intersection of Tup_X and Tup_Y yields the tuples of the constraint. Each pair constructed by matching a value from App_X with a value from d_Y is not a member of Tup_X with probability $P_{\text{constriction}}$. In addition, all pairs constructed by matching values not in App_X to values from d_Y are members of Tup_X. Tup_Y is constructed symmetrically.

High values for $P_{\text{constriction}}$ imply tight constraints.

$P_{\text{applicability}}$ and $P_{\text{constriction}}$ are different from the conventional tightness parameter. They were used instead because they break the artificial, uniform "constrainedness" that tends to occur in constraints generated using the tightness parameter conventionally used to randomly generate binary CSPs. The parameters allow the creation of constraints with more realistic, non-uniform tuple distributions. Many real constraints are in fact "piecewise constraints" ([92]), which affect subsets of their variables' domains differently. The two parameters tend to generate such constraints (each variable domain is partitioned into two sets, one of which is more tightly constrained than the other). Yet, if desired, problems with a uniform distribution of tuples can still be generated by setting $P_{\text{applicability}}$ to zero, and using $P_{\text{constriction}}$ to act as a substitute for the conventional tightness parameter.

The algorithms were implemented in the ECL^{i}PS^e Constraint Logic Programming environment [49], which stores only modified data structures on a backtrack-able stack. Each possible combination of problem parameters was tested 100 times. All trials were conducted on a Sun Sparc Station 20 (100MHz) with 100 CPU seconds timeout per trial. Timeouts were not penalized, but assumed to terminate at the timeout. The majority of timeouts were caused by FC, implying that the results presented for FC appear somewhat more favourable than justified. Nevertheless,

the results still lead to clear conclusions.

4.5.2 Results

Figures 4.7–4.11 compare AC, FC and AFR in terms of average execution time in CPU seconds.

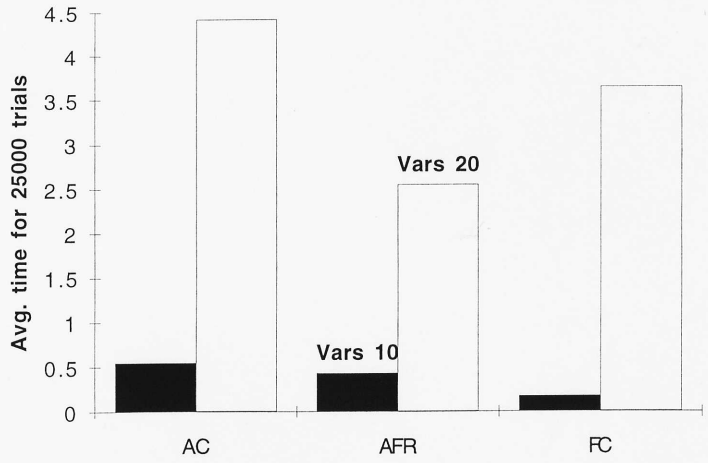


Figure 4.7: Average time (25,000 trials) versus no. of variables

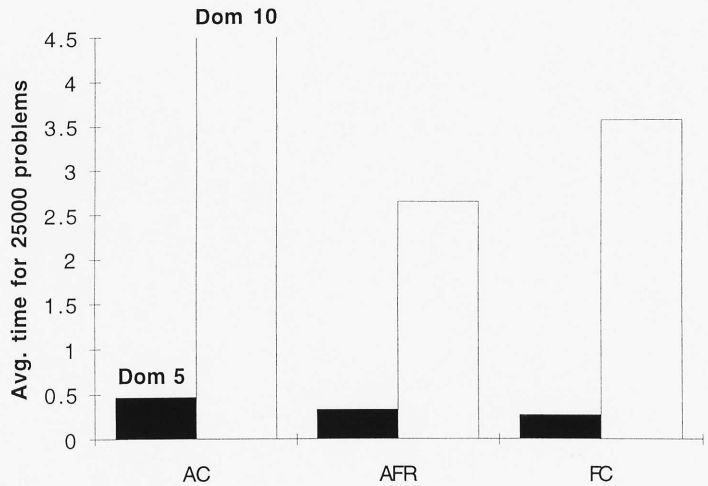


Figure 4.8: Average time (25,000 trials) versus domain size

Figure 4.7 shows that when the number of variables is small (10) the best performing algorithm is FC followed by AFR and then AC. This is consistent with our knowledge

that the application of sophisticated consistency methods is not justified for small problems. However, when we increase the number of variables (to 20) we see an immediate degradation in the performance of FC. The performance of AC and FC become comparable, with AFR outperforming both. A similar effect is seen when domain sizes are increased from 5 to 10 (Fig. 4.8).

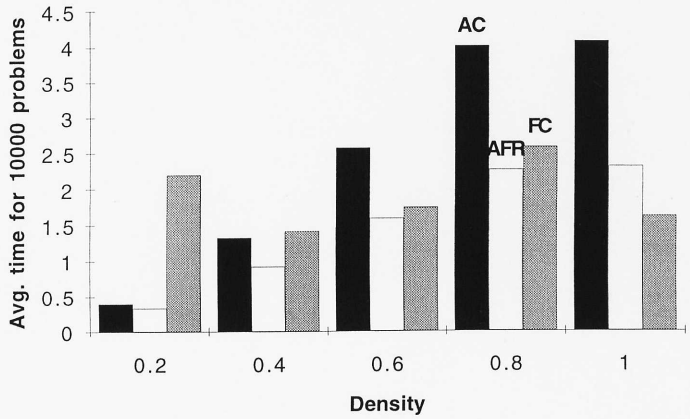


Figure 4.9: Average time (10,000 trials) versus P_{density}

Figure 4.9 demonstrates that AFR improves AC performance across all densities, and is generally better than FC for all but the most constrained problems, with density 1. These problems are usually infeasible and FC detects infeasibility faster because it has lower overheads. Notice though that when AFR is not the best, its performance remains close to that of FC.

The graph for $P_{\text{applicability}}$ in Fig. 4.10 shows again that AFR is superior to AC, and is better than or close to FC performance. Note in particular that AFR improves AC performance for low values of $P_{\text{applicability}}$. This suggests that the type of problems created by a conventional tightness parameter ($P_{\text{applicability}} = 0$) would also lead to clearcut improvements in performance over AC. Note also that high values for $P_{\text{applicability}}$ put FC at a disadvantage; most of the FC timeouts occurred for high values of this parameter. Earlier work has shown that sparseness in a constraint graph can lead to "exceptionally hard problems", for which FC is not effective [84]. While this work relates to density not $P_{\text{applicability}}$, higher values for $P_{\text{applicability}}$ seem

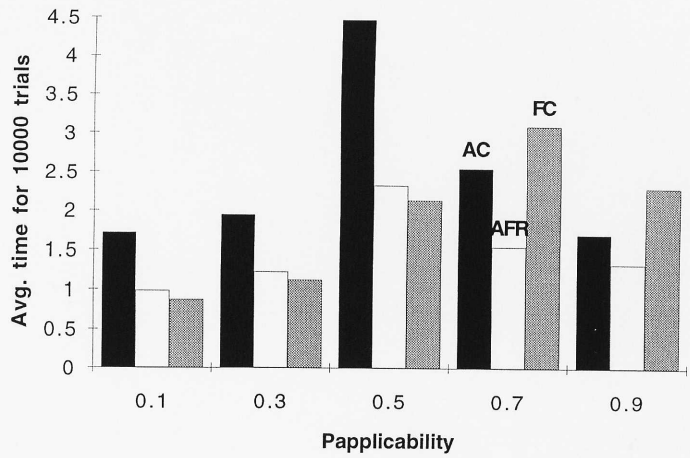


Figure 4.10: Average time (10,000 trials) versus $P_{\text{applicability}}$

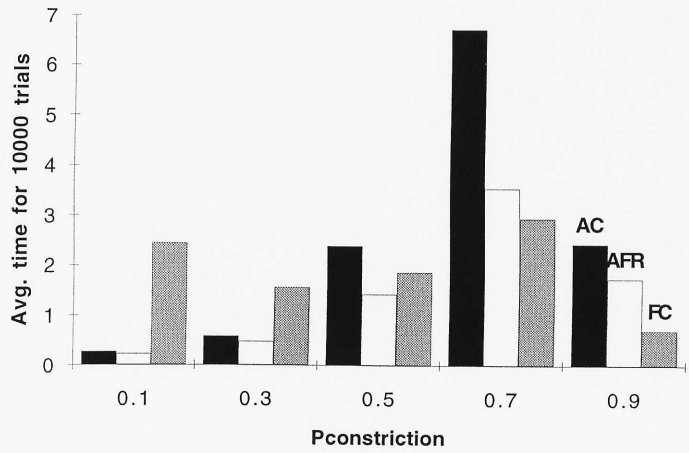


Figure 4.11: Average time (10,000 trials) versus $P_{\text{constriction}}$

to be replicating this effect, because the binary constraints of the problem tend to affect smaller domain subsets.

Figure 4.11, for $P_{\text{constriction}}$, is very similar to the graph for P_{density}, demonstrating that AFR obtains an overall improvement over both AC and FC — despite FC performing better for infeasible problems, where FC's overheads are smaller.

Finally, Fig. 4.12 shows that AFR achieves its best improvements over AC when solving hard problems with a large number of backtracks. Each point marked on the graph represents the average of n hundred trials, where normally $n = 1$ ($n > 1$ when

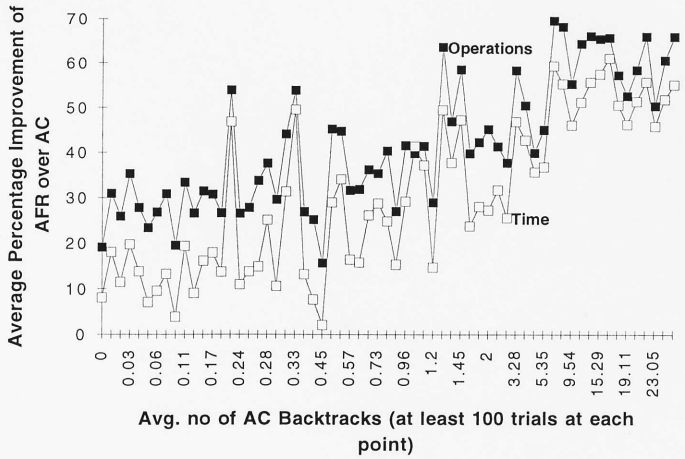


Figure 4.12: Percentage improvement in time/operations versus average backtracks

several trial sets have the same average no. of backtracks). The AFR improvement was measured both in the average absolute time and in the average number of restorable operations; the figure demonstrates the close relationship between the two and shows that AFR reduces AC timings by up to 60% for hard problems, involving many AC backtracks.

4.6 Conclusion

This case study focused on the hybridization of two propagation methods, AC and FC. The adaptive propagation scheme AAP was defined to allow the dynamic scoping of these two consistency methods. An instance of this scheme, AFR, was detailed. AFR is an extended version of the AC-4 algorithm, where the scope of AC propagation is dynamically reduced to exclude arcs that are handled more effectively by FC. Experimental evidence confirmed that AFR performed better overall than pure AC-4 and pure FC. Its performance advantage over AC-4 was greatest on difficult problems involving many backtracks.

In conclusion, this example demonstrates that adaptive propagation can be a viable

and effective means for improving performance. Propagation may be adapted during backtrack search by dynamically scoping different consistency methods to take into account up-to-date meta-knowledge.

Chapter 7 will discuss further the hybridization form applied in this case study, and its wider relevance.

CASE STUDY II

Chapter 5

Sub-Problem Delegation

An early version of this work was published in [25].

5.1 Introduction

5.1.1 Overview

This case study investigates CP algorithms for an important airline LSCO, **fleet assignment**. The case study exposes some of the weaknesses of backtrack search on this problem, and addresses them by hybridizing *forward search* methods using the second hybridization form, **sub-problem delegation.** The forward search methods hybridized in this way are repair-based, and yet complete. On the largest problems tested, these forward search methods yield a significant improvement in algorithm performance.

Fleet assignment is a process performed in airlines ahead of the publication of their schedule. The basic fleet assignment problem involves the selection of aircraft to

perform flights in the schedule. An airline's aircraft are normally grouped into sets of aircraft of the same type, called fleets. The task in fleet assignment is to assign a single aircraft type (a fleet) to every flight in the schedule. The assignments made must respect the number of aircraft in each fleet. As well as being feasible, the assignments must be of good quality. The profitability of each flight is dependent on the fleet assigned to it.

The problem that will be addressed in this case study is in fact a variant of this basic fleet assignment problem. As well as requiring that solutions are relatively profitable, we additionally require that solutions are local improvements of the airline's current fleet assignment. This is because an airline can experience serious disruption if it attempts to implement a new fleet assignment that is radically different from the current one.

We first solve this variant using a backtrack search algorithm together with local consistency methods and search ordering heuristics. Despite the investigation of many combinations of search heuristics, the optimization results are not satisfactory. As we shall see, the difficulty lies in the large size of the optimization search space. Typically, CP backtracking fails to complete the optimization process. Since the search space is visited in a systematic manner, those solutions that are found by the algorithm (before it is aborted) tend to be closely related, and are therefore not fully representative of the problem's solution set. Moreover, local search ordering heuristics can lead the backtrack search to parts of the search space with only poor quality solutions. Often, choices that seem good locally negatively impact a schedule's overall quality, because the problem's variables are highly inter-connected.

This chapter then describes two hybrid CP algorithms that can improve CP backtrack search performance on this problem. The two hybrid CP algorithms apply the same hybridization technique, which involves the application of repair-based, forward search methods to aid backtracking. The hybrids are theoretically sound and

complete, although in practice compromises such as time-outs are used to improve performance. Both algorithms are found to be much more effective than ordinary backtrack search on two large commercial fleet assignment instances.[1]

5.1.2 Fleet Assignment Background

Most research on fleet assignment has focused on the use of mathematical programming methods. Abara provided the first complete description of the problem [1]. In the model proposed, the decision variables corresponded to pairs of flights that could be performed in succession by a single aircraft. Since the number of such pairs is often very large, Abara deployed a heuristic procedure for eliminating unlikely pairs, before applying an MIP-style branch and bound search. Subramanian et al. subsequently reduced the size of the problem by assigning fleets to single flights [85]. They also suggested a pre-processing procedure that heuristically fixes down values for some of the MIP variables. It was claimed that the resulting algorithm might save Delta Air Lines $300 million over three years. Hane et al. studied a similar model and solution method [42]. The models used by Subramanian and Hane et al. are related to each other and to the basic fleet assignment model described here, which was first published in [25].

However, criticisms have been directed at the basic fleet assignment model and the strategies proposed for solving it. The first criticism has been *the omission or dilution of many operational considerations such as crew and maintenance.* Unfortunately, while mathematical programming search strategies are very efficient on the basic problem, their efficiency degrades sharply when such "side constraints" are added. Nevertheless, some work has aimed to extend their capability beyond the basic problem. Clarke et al. investigated mechanisms that allowed the addi-

[1]The commercial fleet assignment problems used in this case study were all supplied by our collaborating airline, British Airways.

tion of operational constraints "without destroying (the fleet assignment problem's) solvability" [11]. The paper suggested specialized model extensions, constraint relaxations, and heuristic techniques to facilitate the approximation of operational constraints. More recently, a new *flight strings* model was proposed to allow such constraints to be handled fully [3]. In this model, every potential flight sequence between two hub airports is represented by a Boolean variable. This extensional representation, in combination with pre-processing, was seen as the only way to fully satisfy operational constraints before solution by a mathematical solver. There can be millions of such decision variables, even in moderately-sized fleet assignment problems. A special mathematical method involving *column and cut generation* was proposed to address the huge scale of the problem. So far, this method has been proved effective only on small fleet assignment problems.

Yet, in our experience, an even more fundamental criticism has been levelled at the basic fleet assignment model: *the optimization function focuses purely on airline profit.* So-called "optimal" assignments have been found by our collaborating airline to be inoperable. The reason is that optimal solutions bear little or no relationship to an airline's current fleet assignment. For example, the optimal solution to the largest fleet assignment problem investigated later, **P-4**, would unrealistically require the airline to change 65% of its flights. Implementing such an assignment would cause unacceptable levels of disruption to the airline's day-to-day operations. In addition, optimal solutions were not sufficiently "robust": they were thought difficult to rebalance on the day of operation after aircraft failures or delays. While robustness is a schedule characteristic that is hard to specify and measure, it was argued that an airline's history causes it, over time, to evolve assignments that are inherently robust. To conclude, our collaborating airline required optimization methods that generated solutions close to its current fleet assignment.

In this case study, we diverge from traditional approaches to fleet assignment, by choosing to construct CP algorithms that do not apply mathematical programming

search methods. In our opinion the relative flexibility offered by the CP paradigm can enable the two criticisms listed above to be more readily addressed. Firstly, CP algorithms can be constructed with greater confidence that their performance will scale better as problems are extended to include disjunctive operational constraints. This is by comparison with the mathematical methods listed above, where good performance is dependent on capturing the most of the problem as a linear programming problem. Secondly, local search heuristics can be used to direct search towards profitable solutions that remain close to the airline's current assignment. This pragmatic approach avoids the difficulties of multi-criteria optimization, which would have to be somehow tackled in methods based on MIP.[2]

5.1.3 Outline of the Case Study

Section 5.2 defines the basic fleet assignment problem and its variant, before Section 5.3 details the CSOP used to represent it.

Section 5.4 then presents the backtrack search algorithm specialized for fleet assignment. A detailed discussion of its weaknesses identifies areas for improvement.

The hybridization technique is described in Section 5.5. Section 5.6 follows with descriptions of the two hybrid algorithms. Limited discrepancy search (LDS) is adapted to the problem at hand, and hybridized into the basic backtrack search algorithm to create the first algorithm. The second hybrid is a variation on the first that uses LDS to conduct a local search.

Section 5.7 compares CP backtrack search with the two hybrid algorithms on four commercial problems, before Sections 5.8 & 5.9 discuss related work, and conclude

[2]Specifically, it is unclear how to satisfactorily combine the two criteria (i.e. profitability, and proximity to the current assignment) within the single objective function that drives MIP-based optimization.

the case study.

5.2 The Fleet Assignment Problem

5.2.1 Description

Basic fleet assignment involves the assignment of aircraft types to a fixed set of flights making up one day of an airline's schedule. Each flight departs from an origin airport at a specified time. The destination airport is also fixed, but the time of arrival is dependent on the aircraft type chosen, since some aircraft are slower than others. Operational constraints can mean that certain flights cannot be assigned to particular aircraft types. Naturally, the aircraft type assigned to a flight must be one of those available for take-off at the origin airport, at the time of departure. The cost of each flight depends on the type of aircraft chosen (the cost here represents reduced profit). The global cost function is defined as the sum of all flight costs, and the objective is to minimize it.

Given this view, some details of an airline's operations and costs can be dealt with at a pre-processing stage. For example, airport curfew constraints (which put limits on the times departures and arrivals can take place) can be satisfied at pre-processing, by eliminating assignments that would cause curfew violations (e.g. slow aircraft arriving too late). Similarly, the cost of an assignment is a summation of many pre-processed costs. These include the number of "spilled" potential passengers (which depends on aircraft capacity), and operational costs such as fuel and airport fees.

Any requirements that cannot be handled in the pre-processing step must be captured as constraints on the possible assignments. In the basic problem, the con-

straints are limited to the *fleet size* and *aggregate maintenance* constraints.[3] These constraints respectively dictate, for each aircraft type, the number of available aircraft, and the percentage that *overnight* (i.e. start and end the day) at airports with maintenance facilities.

A toy fleet assignment example is given below.

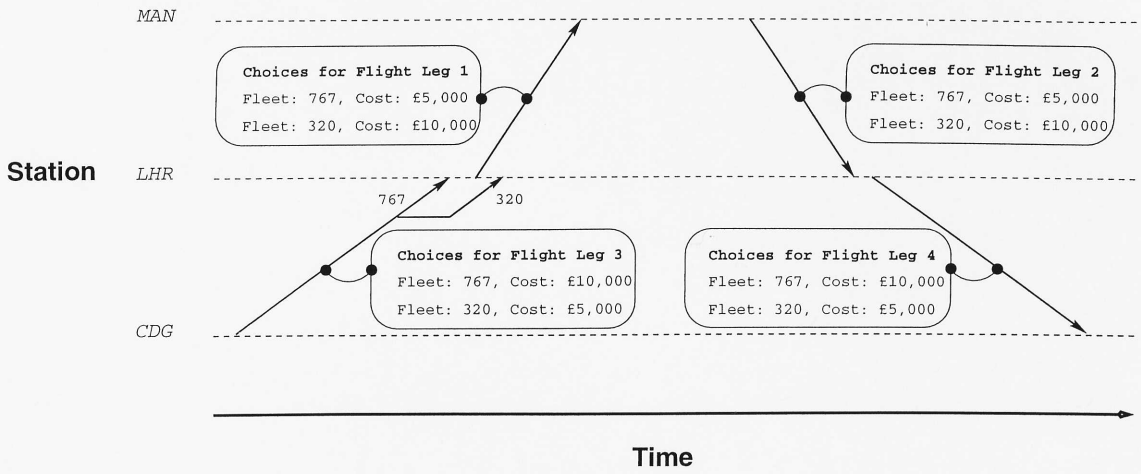


Figure 5.1: An example of a fleet assignment problem

EXAMPLE : Figure 5.1 illustrates the toy fleet assignment problem.[4] Three stations (airports), MAN, LHR and CDG are serviced by four flights. Each flight is performed by one of two plane types, a Boeing 767 or an Airbus 320. Every choice is associated with a pre-processed cost. If a 767 is chosen for one of the flights (Flight Leg 3), it will be ready to undertake the next flight earlier than if a 320 is chosen. Clearly this has implications for the number of aircraft required to perform the schedule. For instance, if a 767 is chosen to perform all the flights, only one aircraft is needed. In contrast, choosing only 320s would require two aircraft, because the one performing Flight Leg 3 will be ready only after Flight Leg 1 departs. The optimal assignment

[3]These constraints are common to the basic problem as defined for Delta Airlines, US Air and British Airways.

[4]All details of this example are fictitious.

for this problem uses a 767 for Flight Legs 1 and 2, and a 320 for Flight Legs 3 and 4. However, the optimal cost of £20,000 must be weighed against the consideration that, again, two aircraft are needed to execute the schedule. □

5.2.2 Definition

The problem components

The fleet assignment problem is based on three components:

- A set of stations S

  A station $s \in S$ represents an airport or several adjacent terminals at an airport. The relation $maint(s)$ holds iff s is a maintenance station.

- A set of plane types P

  Each plane type $p \in P$ is associated with a fleet size $fs(p)$, and a required nightly maintenance percentage $mpc(p)$.

- A set of flight legs L

  Each leg $l \in L$ has an origin $o(l)$ and a destination $d(l)$, as well as a fixed departure time $t_{dep}(l) \in \mathbb{T}$. A Plane Type variable pt_l represents the plane type assigned to the leg l, and is associated with a domain $d_{pt_l)} \subseteq P$. Every flight and plane type pair is associated with an assignment cost $c(l, p) \in \mathbb{N}$, and a ready time $t_{ready}(l, p) \in \mathbb{T}$. If a particular plane type p is assigned to a l, $t_{ready}(l, p)$ defines when the assigned plane will be ready to undertake the next flight.

Feasibility

For each pair $(s, p) : s \in S, p \in P$, let an overnight variable on_{sp} signify the number of aircraft of plane type p overnighting at s, *before* the schedule is executed. Since the distribution of aircraft must return to its previous state at the end of the day, on_{sp} is also equal to the number overnighting *after* the schedule is executed. Let ON be the set of all overnight variables.

A *feasible solution* is an assignment to all the Plane Type variables $PT = \{pt_l : l \in L\}$ and the overnight variables ON such that four types of constraints are satisfied. The four constraint types are explained below, before they are formally defined.

i. The *fleet size constraint*, which ensures that, for each plane type, the total number of aircraft overnighting at the stations is equal to the size of the fleet.

ii. The *aggregate maintenance constraint* which, for each plane type, forces the total number of aircraft overnighting at maintenance stations to be greater than or equal to the required maintenance percentage.

iii. The *balanced schedule* constraint, which specifies that, at each station, the number of departures assigned to a given plane type must be equal to the number of arrivals assigned to that type.

iv. The *aircraft usage* constraint, which ensures that enough aircraft of each plane type overnight at the stations. At any point in time, the number of overnighting aircraft must be at least as great as the **aircraft deficit**. The aircraft deficit is defined, for every point in time, to be the number of departures prior to that time point minus the number of arrivals prior to that time point.

Formally, these constraints are defined as follows:

$$
\begin{array}{lrcl}
\text{(i)} & \forall p \in P : \sum_{s \in S} on_{sp} & = & fs(p) \\
\text{(ii)} & \forall p \in P : \sum_{s \in S \wedge \text{maint}(s)} on_{sp} & \geq & mpc(p) \cdot fs(p) \\
\text{(iii)} & \forall s \in S, p \in P : |\{l : o(l) = s \wedge pt_l = p\}| & = & |\{l : d(l) = s \wedge pt_l = p\}| \\
\text{(iv)} & \forall s \in S, p \in P, t \in \mathbb{T} : on_{sp} & \geq & \Delta(s,p,t)
\end{array}
$$

where $\Delta(s,p,t)$ represents the aircraft deficit (as defined above) at station s of type p at time t:

$$
\Delta(s,p,t) = \left| \left\{ l : \begin{array}{c} pt_l = p \\ o(l) = s \\ t_{dep}(l) \leq t \end{array} \right\} \right| - \left| \left\{ l : \begin{array}{c} pt_l = p \\ d(l) = s \\ t_{ready}(l,p) \leq t \end{array} \right\} \right|
$$

Optimality in the basic fleet assignment problem and its variant

Finally, in the basic fleet assignment problem the *optimal solution* is a feasible solution minimizing the total cost $\sum_{l \in L} c(l, pt_l)$.

However, as stated earlier, there exists a second "implicit" minimization criterion: the degree of change to the airline's current fleet assignment. The value of this second criterion can be measured by $|\{l : pt_l \neq current(l)\}|$ where *current* is a function mapping each flight leg $l \in L$ to the fleet it is currently assigned to. However, the algorithms presented here do not explicitly optimize this second criterion. The algorithms all search for solutions that are optimal with respect to the first criterion, using the second criterion only to influence the search ordering.

5.3 Representing Fleet Assignment as a CSOP

This section describes a CSOP model for fleet assignment. In the problem formulation given above, constraints (iii) and (iv) are not directly expressible using simple numeric constraints. To model the problem using numeric constraints two sets of peripheral variables are introduced into the CSOP: Boolean Plane Type variables, and Delta variables.

Boolean Plane Type Variables A 0-1 Boolean is introduced for each pair of flight leg $l \in L$ and plane type $p \in P$, to represent whether l is assigned to p:

$$\forall l \in L, p \in P : b_{lp} = 1 \Leftrightarrow pt_l = p$$

Constraints on quantities of aircraft of particular plane types can be directly defined in terms of these Boolean variables.

Delta Variables As stated, constraint (iv) constrains the aircraft deficit $\Delta(s, p, t)$ for all stations, plane types and time-points. An enhanced representation is more practical and efficient; changes only occur to the $\Delta(s, p, t)$ after departure and arrival time-points from s. In addition, since only the upper bound is constrained, we need only observe the value of $\Delta(s, p, t)$ at departure time-points $\{t|\exists l : t = t_{dep}(l) \wedge o(l) = s\}$. In this view, the time line is divided into adjacent intervals separated by departure times. At station s, the critical time-points reduce to:

$$T(s) = \{t_{start}\} \cup \{t|\exists l : t = t_{dep}(l) \wedge o(l) = s\} \cup \{t_{end}\}$$

where t_{start} and t_{end} are the time points before and after which no flights depart or arrive at s.

The value of $\Delta(s,p,t)$ may now be constructed incrementally left-to-right on the time line. A delta variable Δ_{spt} corresponding to the value for $\Delta(s,p,t)$ is introduced for each $s \in S, p \in P, t \in T(s)$. The delta variables are related to each other, and constrained to be always less than the number of overnighting aircraft (ensuring correct *flow logic* in the new schedule). By forcing the deficits at t_{start} and t_{end} to be zero, the number of departures of a plane type is made equal to the number of arrivals of the same type over the schedule horizon.

Let $I(s)$ be the set of time line intervals $(t_1, t_2]$ such that t_1 immediately precedes t_2 in an ascending ordering of $T(s)$. We now reformulate constraints (iii) and (iv) as follows:

Constraint (iii): $\forall s \in S, p \in P : \Delta_{spt_{start}} = \Delta_{spt_{end}} = 0$

Constraint (iv): $\forall s \in S, p \in P : \forall (t_1, t_2] \in I(s) :$

$$\Delta_{spt_1} + \sum_{o(l)=s \wedge t_{dep}(l)=t_2} b_{lp} - \sum_{d(l)=s \wedge t_1 < t_{ready}(l,p) \leq t_2} b_{lp} = \Delta_{spt_2}$$

$$\forall s \in S, p \in P : \forall t \in T(s) : \Delta_{spt} \leq on_{sp}$$

5.4 CP Backtrack Search (BT)

This section details the backtrack search algorithm developed for the fleet assignment variant problem. The backtrack algorithm is first enhanced conventionally, via the application of local consistency techniques, and the specialization of search heuristics. The aim is to generate a good quality backtrack algorithm, which serves a number of different purposes:

- It indicates the levels of performance that can be expected from a conventional, specialized CP backtrack search algorithm. This gives us a yardstick by which to measure other hybrid algorithms.
- It allows us to identify the sources of its weak performance, and thus focus better the hybridization in subsequent sections.
- It provides the core of the hybrid algorithms, namely the backtrack search component, which will be aided by support methods.

5.4.1 Overview of Algorithm

The algorithm conducts search by labelling the Plane Type variables of PT (the Boolean Plane Type Variables are made to take corresponding values). It applies arc-B consistency propagation on the arithmetic constraints of the CSOP [58]. A branch and bound optimization framework is applied to ensure that returned solutions are always better than those previously found.

5.4.2 Enhancing the Algorithm

A significant effort was invested in terms of problem analysis and experimentation to improve algorithm performance. This led us to identify useful classes of redundant constraints, and search ordering heuristics. These are described below.

Redundant Constraints

Two sets of redundant constraints are used to improve the efficiency of search. The first set of redundant constraints lead to the removal of non-essential variables, while

the second set can reduce the number of search nodes visited, by achieving earlier prunes of the PT decision variables:

Zero-capacity constraints:

If a station is at Zero-Capacity (has no available aircraft) at two time-points in the day, then it is possible to deduce that the set of incoming aircraft is identical to the set of outgoing aircraft between the two time-points. If, between two zero-capacity points, only one arrival l_1 is followed by only one departure l_2, their Plane Type variables may be unified during pre-processing: $pt_{l_1} = pt_{l_2}$ and also $\forall p \in P \cdot b_{l_1 p} = b_{l_2 p}$. Since this situation often occurs in practice, Zero-Capacity constraints remove many redundant variables in a pre-processing step. This enables space savings and reductions in arc-B consistency checking.

Total-delta constraints:

These redundant constraints are produced by reasoning over all the incoming and outgoing flights at a station. The reasoning is across all fleets, unlike the flow reasoning of constraint (iv). At every departure time it is possible to calculate the *total* aircraft "deficit". This is the difference between the total number of departures and the total number of arrivals up to that time.[5] At each departure time, a redundant constraint is added to ensure that the sum of the separate plane type deficits is equal to this total deficit. The constraints can lead to tighter domain bounds for the plane type deficits, and thus to reductions in the number of search nodes visited.

The following example illustrates one instance of this type of reasoning:

EXAMPLE : Let a station $s \in S$ have zero overnighting aircraft. If the station is at zero-capacity at mid-day (t_{mid}) because the total number of arrivals equals

[5]Some flights may arrive either *before or after* a departure time, depending on the plane-type assigned to perform them. In these situations, the total deficit is a range of values that gets fixed only when the flights are assigned.

the total number of departures up to that time, we can infer immediately that all the separate plane type deficits at mid-day are exactly zero ($\forall p \in P : \Delta_{spt_{mid}} = 0$). Unless the redundant constraint is included, this is not immediately inferred. □

Search Ordering

The decision variables of the fleet assignment CSOP are the flight Plane Type variables $PT = \{pt_l : l \in L\}$. In the largest problem studied later, **P-4**, there are almost 400 such variables (flights), with a maximum domain size of six (the number of fleets). Most of the remaining 7500 secondary variables (Booleans, deficit variables, overnights, etc.) are *functionally dependent* variables that are forced to take unique values once the Plane Type variables are assigned. The rest are the overnight variables ON, which are not strictly functionally dependent, but do not require additional search: arc-B consistency will ensure they may be consistently fixed to their minimum bounds, after the Plane Type variables have been labelled.

The following variable and value ordering heuristics were compared on **P-4**, to identify the best combinations:

Variable ordering heuristics:

1. *Station Type and Size.* As an optional primary heuristic, select first a station, ordered either by maintenance capability and/or size, then prefer the variables departing from or arriving at that particular station. One of the other heuristics is then applied as a secondary heuristic.
2. *Earliest Departure/Arrival.* As a secondary heuristic, choose the Plane Type variable that departs or might arrive next.
3. *Minimal Domain Size (MDF).* Choose the Plane Type variable with the smallest domain.

Value ordering heuristics:

1. *Current Assignment.* Attempt the airline's current assignment first if possible, using one of the other heuristics otherwise.
2. *Largest Fleet.* Choose the plane type of the largest fleet first.
3. *Best Cost.* Choose the plane type with the best cost for the flight.
4. *Available Aircraft.* Choose the plane type of the fleet with the greatest number of "available" aircraft. As labelling progresses, the quantities of aircraft known to overnight at each station gradually rise. This dynamic ordering heuristic prefers the fleet with the greatest number of uncommitted planes.

Search ordering results. All combinations of the above heuristics were attempted on the largest problem, **P-4**. The best performing variable ordering heuristic was found to be Maintenance Station first, Ascending Station Size, combined with Earliest Departure/Arrival first variable ordering. The primary value ordering heuristic was Current Assignment first, which was applied to prefer solutions that are close to the airline's current assignment, as required in our fleet assignment variant. The best performing secondary heuristic was the Largest Fleet first. This combination of heuristics is applied in the experimental comparisons of Section 5.7, both for the pure and the hybridized backtrack algorithms.

5.4.3 Backtrack Algorithm Pitfalls

Experiments on the benchmark problems described later show that the CP backtrack search algorithm is capable of producing fleet assignments that are better than an airline's current assignment, and yet similar to it. However, its performance leaves much room for improvement. We describe here the principal reasons for its weak performance. These reasons are used to identify opportunities for hybridization.

The core problem of backtrack search is the fact that decisions made early on, close to the root of the search tree, are very strongly committed to. The algorithm only reconsiders them after exhaustively searching large parts of the search space.

For large-scale problems where the algorithm does not execute to completion, this leads to *a skewed sampling of the solution set.* Early variable assignments close to the search tree root are not changed before the algorithm is aborted. Since all the generated solutions include these early assignments, they are not fully representative of the solution set.

Moreover, *the early decisions can be of poor quality, because either:*

1. *they are inconsistent with each other, and this is not detected by the local consistency checking, causing the backtrack search to thrash;*
2. *or they direct search to regions where the solutions are not good with respect to the optimization function.*

We consider briefly how such situations can arise in the context of fleet assignment.[6] The problem is that decision variables (plane type variables) labelled during the early stages of search rarely cause reductions of the domains of other decision variables. Yet, they may have global consequences that only become apparent as the search progresses. We use a simple example to illustrate this pitfall.

EXAMPLE : Suppose that the backtrack search has assigned a Boeing 767 to execute flight XX100 from CDG to MAN departing at 10:00 and arriving at 11:00. This 767 is subsequently unable to perform any flights that cannot connect with flight XX100 — for instance, flight XX200 departing from LHR at 11:30. However the plane type variable of flight XX200 is not necessarily pruned, because alternative

[6] A more detailed analysis of the sources of algorithm weakness is given in [25].

767s could still be available to execute it. Yet the fact remains that the number of 767s available to execute XX200 has been reduced by 1. This may be problematic if a 767 is the preferred plane type for flight XX200, since it now has to compete for the *reduced* set of 767s with any other flights that prefer or need a 767. □

Thus early assignments may impact global feasibility and cost. However their effects may become apparent *only after many other variables have been labelled.* At this stage, if those early assignments cannot be extended to solutions, the backtrack algorithm will thrash, or, if they have a negative impact on global cost, the algorithm will generate solutions of poor quality.

It appears then that labelling a plane type variable in isolation is a decision that is too specific and low-level, with wide consequences that are not exposed, at least by our algorithm's search heuristics and local consistency checking. In order to improve the performance of the algorithm, two alternatives exist. The first alternative is to enhance the constraint model, the search heuristics and/or the local consistency checking methods. This can be a specialized and time-consuming task, where the ultimate objective is to avoid making any bad choices at all, at least in the early stages of search. The second alternative is to alter the structure of the search to make it more "forgiving" of such bad choices. This option is the one pursued in the following section.

5.5 The Hybridization Technique

We now proceed to the hybridization technique that is applied to enhance the performance of the backtrack search algorithm. First, a description is given for the generic hybridization form that underlies the technique. Then the form is specialized for our fleet assignment problem.

The hybridization technique draws upon several ideas from earlier work. Later, Section 5.8 will discuss the research that aided its development.

5.5.1 Sub-Problem Delegation — the Hybridization Form

As explained in Chapter 1, the term "hybridization form" describes a hybridization technique that is general enough to hybridize more than just one set of methods. We introduce here sub-problem delegation, the first of two generic hybridization forms that will be used in this thesis to integrate a forward search method into the backtrack search algorithm (the second is detailed in Case Study III).

The form utilized here works by consigning parts of the problem — that are not well-suited to backtracking — to the forward search method. At some point during the backtrack search, a subset of the unassigned problem variables are delegated to the forward search method. Henceforth, the forward search is responsible for labelling these variables. The forward search method activates its own independent search to obtain labels for the variables it has been allocated. When these are found, the backtrack search applies them and then searches the remainder of the problem. On backtracking, the forward search method is invoked again to return an alternative set of labels for the variables.

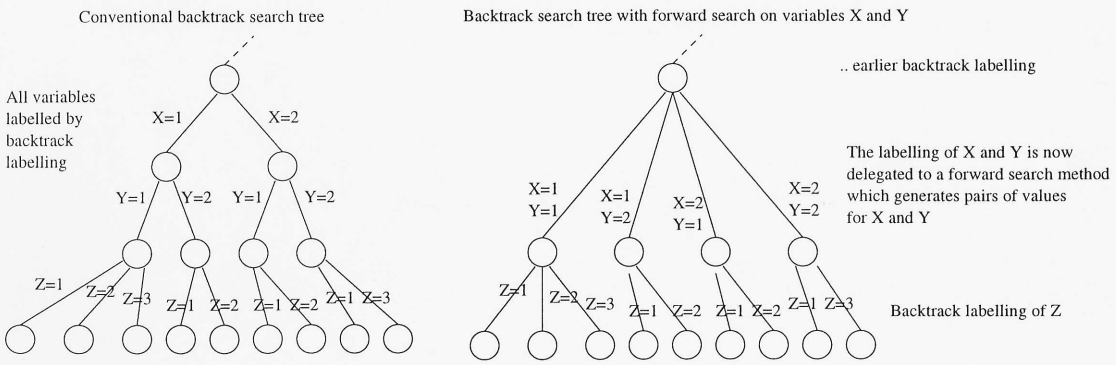


Figure 5.2: An example of the hybridization form

EXAMPLE : Figure 5.2 compares two search trees for solving a problem involving three variables X, Y and Z. The search tree on the left shows a conventional backtrack search tree, where each search node is associated with a choice of alternative assignments to a variable. The search tree on the right shows the same search, but now the task of labelling variables X and Y is delegated to a forward search method. At the top of the search tree, the forward search method is invoked for the first time, to return the first set of values for X and Y (X=1 and Y=1). These labels are used together by the backtrack search, which then continues as normal, labelling Z to several different values. On backtracking, X and Y are labelled again by the forward search method to a different set of values, before control returns again to the backtrack search. The search continues until no further assignments for X and Y are possible. □

Thus, whenever forward search is invoked, the partial assignment associated with a backtrack search node is extended by several variable labels in one step. These labels are later changed by the forward search method if a backtrack occurs. To preserve algorithm soundness, the labels for the variables must always be consistent extensions of the search node's partial assignment. To guarantee completeness, the forward search method must be capable of generating (one at a time) all the labels that can be extended to a solution.

5.5.2 The Form Specialized for Fleet Assignment

We return now to our fleet assignment problem. The discussion of the pitfalls of the backtrack search suggested two ways in which efficiency could be improved. First, early decisions should be made at a higher level than single plane-type assignments, and have consequences that are more easily exposed by the backtrack algorithm's search heuristics and local consistency checking. Second, such early decisions should be made more easy to revise.

To achieve these two objectives, we apply a forward search method at the start of the search, using the hybridization form described. The forward search method is delegated the task of labelling the overnight (ON) variables. These variables are labelled before any plane type (PT) variables are fixed by backtrack search. In labelling the overnight variables, the forward search method distributes the available aircraft over the stations of the problem. The resulting problem is found to be much more amenable to solution by backtrack search.

The forward search method is a repair-based search algorithm. This is because of the requirement to stay close to the airline's current assignment. At the start of search, the distributions of fleets over stations in the current assignment is applied in one step to the overnight variables. Upon backtracking, the method perturbs this distribution, finding new neighbours.

The method thus partitions the problem into a number of relatively (but not fully) independent sub-problems. Each sub-problem corresponds to a station $s \in S$ of the fleet assignment problem. The sub-problems are still interconnected because each plane type variable remains relevant to both its departure and arrival stations.

Thus the assignments generated by the forward search method for the overnight ON variables focus the subsequent backtrack search on a much tighter, and less connected problem, to which it is better suited. Once this reduced problem has been searched exhaustively (or a time-out stops the backtrack procedure) backtracking occurs and control is returned again to the repair-based method. A new assignment of the overnight variables is then generated, creating an alternative reduced problem. As before, this is then searched by backtrack search. In this way, the search interleaves the two methods until the space of repairs is exhaustively searched.

5.5.3 The Hybridization Pseudo-Code

Figure 5.3 gives the high-level pseudo-code for the hybridization technique. It consists of a procedure that takes as its arguments the current assignment CurrentSoln; the overnight variable-domain pairs ONVarDoms; all other such pairs OtherVarDoms; and finally the set of problem constraints Constrs. The procedure overnight_values returns the values of the overnight variables according to the current assignment. repair_init then initializes the repair algorithm with the problem parameters. OtherVarDoms and Constrs are included because the repair algorithm uses consistency checking to enhance its performance. The algorithm then starts the repair iterations, by calling backtrack search with the overnight variables fixed. In the two hybrids presented a time-out will always be applied to this backtrack search call. The call returns either a solution, which is added to the set of accumulated solutions, or $\perp$ indicating none has been found. If a solution is found the repair method is notified by a call to repair_notify_soln_found() (the behaviour of the repair algorithm can depend on whether a solution is successfully returned by backtrack). repair_get_next_neighbour() then returns the next set of values for the overnight variables, as generated by the repair algorithm. This process continues until repair_get_next_neighbour() indicates no further repairs are possible by returning $\perp$.

In the algorithms, the entire fleet_hybrid procedure is applied as part of a branch and bound optimization process: solutions are only generated when they are better than those previously found.

5.5.4 Summary

Notice the three main advantages of the hybridization technique:

1. The technique delegates early search decisions — the point where backtrack

```
1.  begin fleet_hybrid(CurrentSoln, ONVarDoms, OtherVarDoms, Constrs)
2.      ONAssignment := overnight_values(CurrentSoln)
3.      repair_init(ONAssignment, ONVarDoms, OtherVarDoms, Constrs)
4.      Solns := ∅
5.      while ONAssignment ≠⊥ do
6.          Result:= backtrack(ONAssignment, OtherVarDoms, Constrs)
7.          if Result ≠⊥ then
8.              Solns:= Solns ∪ Result
9.              repair_notify_soln_found()
10.         ONAssignment := repair_get_next_neighbour()
11.     return Solns
12. end fleet_hybrid
```

Figure 5.3: fleet_hybrid: A hybridization technique for fleet assignment

search's commitments are strongest — to the forward search method. This allows poor-quality decisions to be more easily modified at a later stage.

2. By starting with an overnight distribution equivalent to that of the airline's current fleet assignment, and then searching those distributions that are its "neighbours", the forward search method gives priority to the sub-problems in the vicinity of the airline's assignment.

3. Within each sub-problem, the backtrack procedure is made to search a much reduced problem. Even if the backtrack search procedure times out, the plane type PT assignments made close to the root of the backtrack search tree are more likely to have been revised. Thus the solutions are less skewed, and the technique generates more representative solutions from the space around the airline's assignment.

5.6 The Forward Search Methods

This section presents two forward search methods that are incorporated into hybrid algorithms using the hybridization technique.

Recall that the fourth hybridization condition (Section 3.1) requires that the basic hybrid algorithm is sound and complete. Thus the forward search method must be complete, in order to generate all possible assignments to the overnight variables that can result in a solution. The question is how to conduct a repair-based search that is complete and yet space efficient.

The forward search methods presented both tackle this issue by utilizing limited discrepancy search (LDS).

5.6.1 Limited Discrepancy Search

Limited discrepancy search is an algorithm originally designed to improve the performance of backtrack search when it is enhanced with sophisticated search ordering heuristics. The LDS approach addresses the case when the heuristics fail, and backtracking thrashes. Even high quality heuristics make the wrong choice at some search nodes.

LDS is based on the assumption that the failing heuristic would have succeeded were it not for a few wrong choices, called discrepancies. To counter this, LDS allows the backtrack search to go against the heuristic exactly n times. If a solution involves exactly n discrepancies, it will be found by the backtrack algorithm. n is initially set to zero and incremented, possibly up to the maximum number of choices made. The search terminates whenever a solution is found. Harvey and Ginsberg describe limited discrepancy search in more detail in [44].

A form of LDS can be used to implement a repair-based backtrack search. In "repair-based LDS", the choices made are variable assignments, and the "heuristic" recommends the assignment of variables to their values in the current assignment. The number of discrepancies in this case is the number of variables assigned differently from the current assignment.

However, since the hybridization technique applies the repair-based method to numeric variables (the overnight variables), a more accurate notion of discrepancy is used. The discrepancy n is not the number of variable assignments that are different from those in the current solution. It is the *absolute difference* between the old and the new assignments:

$$n = \sum_{v \in V} abs(v - old(v)) \tag{5.1}$$

where V is the set of variables, and *old* is a function mapping variables to their old values.

5.6.2 Forward Search by LDS

The first hybrid applies a forward search method based on this style of LDS. The current overnight distribution of aircraft is returned first, creating the initial sub-problem. Once backtrack completes (or times-out), it produces new distributions that diverge increasingly from this initial distribution. Each new distribution is the basis of a new sub-problem that is searched by the backtrack algorithm. Eventually all possible overnight distributions will have been attempted, because the resulting search is in principle complete.

The forward search method works by posting the discrepancy expression of Eqn. 5.1 as a constraint, such that the set of variables V comprises the overnight variables. n is fixed to increasing values, starting from zero, before the overnight variables are then labelled by a normal backtrack procedure. During the labelling, the LDS algorithm applies arc-B consistency to *all* the problem's constraints (including constraints on variables that are not labelled by the LDS algorithm).

The application of LDS as a forward search method in the hybridization procedure fleet_hybrid is straightforward. The procedure repair_init sets up the repair-based LDS search on the overnight variables. n is set to zero, fixing the overnight variables to their values according to the current fleet assignment. Since the success or failure of the call to backtrack does not influence LDS, repair_notify_soln_found() is ignored (in contrast to the next method, Hill Climbing LDS). repair_get_next_neighbour() causes LDS to obtain the next set of labels for the overnight variables. n is incremented whenever no new labels are found. The search terminates when it surpasses the maximum possible discrepancy.

In the experimental comparisons, the hybrid algorithm that uses this forward search method is denoted LDS.

5.6.3 Forward Search by Hill-Climbing LDS

In the second forward search method, LDS is harnessed to create a local, and yet complete form of repair-based search. This is achieved by updating the old labels the LDS algorithm is measuring discrepancy from, whenever a new solution is found. The search is thus a hill-climbing one, where moves to a neighbour search node (set of overnight labels) occur whenever it is associated with a solution with better cost. Again, the LDS search is aided by local consistency checking on the problem constraints.

The hill-climbing LDS method is integrated as follows. The method is notified of new solutions by calls to repair_notify_soln_found(). When repair_notify_soln_found() is called, the constraint of Eqn. 5.1 is withdrawn, and replaced by a new constraint where the function *old* returns the values that were last returned by LDS for the overnight variables. The discrepancy number n is reset to 1, and LDS search is restarted. Note that hill-climbing LDS can re-visit already tested overnight distributions, but because time-outs are applied to the calls to backtrack, this behaviour is desirable. Every time a new solution is returned by backtrack, the problem is tightened by a reduced upper bound on cost. This is a mechanism of the branch and bound optimization process, designed to ensure that only better solutions are generated. Hence previously attempted sub-problems are re-visited after more accurate cost bounds are obtained.

Applied to fleet assignment, hill-climbing LDS can improve ordinary LDS because it focuses search on the most promising regions, and can re-attempt, with better cost bounds, sub-problems where a time-out has occurred.

More generally, hill-climbing LDS illustrates how LDS can be used as a generic neighbourhood operator for local search. Such a neighbourhood operator creates local and yet complete search methods, with the low space complexity of ordinary backtrack search. The neighbourhood operator can be made to incorporate any constraints, via consistency checking. However, unlike conventional neighbourhood operators, there is no guarantee that worst-case time complexity will remain polynomial.

The hybrid algorithm integrating this forward search method is referred to in the experimental section as HLDS.

5.7 A Comparison of the Algorithms

The empirical comparison set out to compare the algorithms described on four different fleet assignment problems provided by British Airways, named **P-1** up to **P-4**. Table 5.1 gives the dimensions of the four problems, ordering them according to their complexity (least complex first):

Table 5.1: The fleet assignment problems

Problem	No. Flights	No. Stations	No. Aircraft	No. Fleets	Fleet Sizes
P-1	96	24	20	2	11/9
P-2	98	24	22	3	11/6/5
P-3	244	38	53	4	24/19/6/4
P-4	372	56	76	6	28/22/10/8/6/2

Table 5.2 lists the six different algorithm variants applied to these tests. The algorithms were all implemented on the ECL^{i}PS^e Constraint Logic Programming platform [49]. All algorithms ran on a Sun Ultra Enterprise 2-170 (170MHz), and were stopped after 50,000 CPU secs.

Table 5.2: The fleet assignment algorithms

Algorithm	Description
BT	The backtrack algorithm presented in Section 5.4, applying the heuristics that give the best results on the largest problem, **P-4**.
BT(ON)	A variation on BT where the overnight variables are also used as decision variables, and selected before the plane type variables.
LDS-50	The LDS hybrid algorithm of Section 5.6.2, with a 50 CPU sec. time-out on calls to backtrack.
LDS-500	The LDS hybrid algorithm with a 500 CPU sec. time-out on calls to backtrack.
HLDS-50	The hill-climbing LDS hybrid algorithm of Section 5.6.3, with a 50 CPU sec. time-out on calls to backtrack.
HLDS-500	The hill-climbing LDS hybrid with a 500 CPU sec. time-out on calls to backtrack.

The first two benchmark problems, **P-1** & **P-2** involve very similar sets of approximately 100 flights. **P-1** was an investigation of two fleets on this set of flights. **P-2** differs in one principal respect; one fleet in **P-1** (comprising 9 aircraft) is expanded, and split into two **P-2** fleets (comprising 6 and 5 aircraft), to make a total of three fleets.[7] The problems are of interest since they are relatively small in size, and do not allow a great deal of flexibility in the overnight distribution of aircraft. The lack of flexibility will be seen to reduce the effectiveness of the hybridization technique. The low flexibility is a product of the small number of stations and plane types, and the fact that only one of the stations is a hub/maintenance station.[8] Nevertheless, the additional plane type available in **P-2** means that more aircraft distributions may be attempted than for **P-1**. This increased flexibility will be found to improve the effectiveness of LDS and HLDS.

The algorithms tested on **P-1** & **P-2** were BT, BT(ON), LDS-50 and HLDS-50. Figures 5.4 & 5.5 show the results of these tests. The results describe the optimization performance of BT(ON), LDS-50 and HLDS-50 relative to that of BT. The relative performance was measured, and indexed to CPU time. We introduce the notion of the *cost reduction* of an algorithm. This measures the best reduction in cost obtained, relative to the airline's current assignment, as execution time increases. Specifically, at any point in (CPU) time, it is the difference between the cost of the airline's current solution and the cost of the best solution obtained so far by the algorithm in question.

The cost reduction of an algorithm is normalized by dividing it by the cost reduction of backtracking to create its *cost reduction factor* (CRF). Again, this is indexed to execution time. The CRF function $crf(Alg, t)$ is defined as follows. $\forall Alg \in \{\text{BT(ON)}, \text{LDS-50}, \text{HLDS-50}\}, t \in \mathbb{T}$:

[7]The two fleets were first grouped together in **P-1** because they were considered to be broadly similar.

[8]When there is no choice of maintenance station, a significant proportion of aircraft must overnight at the sole maintenance station, because of the maintenance constraint (Constraint (ii) of Section 5.2.2).

$$crf(Alg, t) = \frac{cost_red(Alg, t)}{cost_red(BT, t)}$$

where $cost_red(Alg, t)$ represents the cost reduction of Alg at time t.

Thus, at any point in time, a CRF below one indicates worse performance than BT, while a value above one indicates better performance.

The columns of Figures 5.4 & 5.5 show the CRF values for BT(ON), LDS-50 and HLDS-50 sampled at various execution times. Columns that are missing indicate that the algorithm in question has terminated during an earlier interval. The column on the right shows the final CRF, found by comparing the best solution obtained by each algorithm with the best obtained by BT at 50,000 CPU secs.

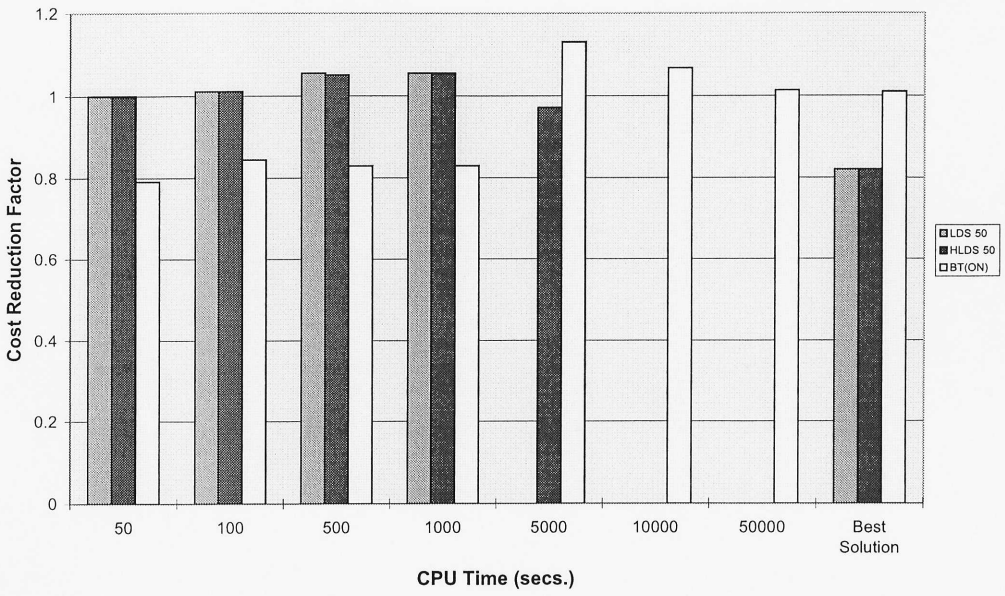


Figure 5.4: Optimization performance on **P-1**

The first figure shows that for **P-1**, LDS-50 and HLDS-50 have similar performance,

and remain broadly in line with BT until they terminate. For LDS-50, termination occurs between 500 and 1000 CPU secs., and for HLDS-50, between 1000 and 5000 CPU secs. The terminations occur because they have exhaustively visited the limited overnight distributions that are possible. With a time-out of 50 secs., the calls to backtrack were unable to return better quality solutions than those already generated, for any of the overnight distributions. However, BT and BT(ON) continue until the final cut-off time of 50,000 CPU secs.; by then they will have obtained better solutions than those found by LDS-50 and HLDS-50 before they terminated. Interestingly, the performance of BT(ON) is no better than that of BT, indicating that simply labelling the overnight variables within backtrack search does not on its own dramatically improve performance.

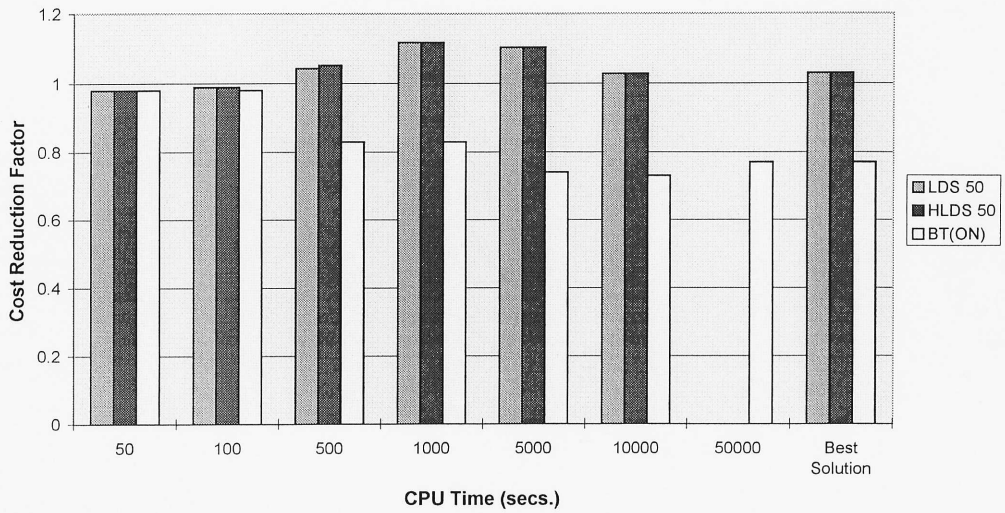


Figure 5.5: Optimization performance on **P-2**

The second figure shows changes to the algorithm CRFs for **P-2**. These occur because of the additional plane type, which increases the number of potential overnight distributions. The additional combinations cause LDS-50 and HLDS-50 to execute

for much longer, now terminating between 5,000 and 10,000 CPU secs. Now their solutions are marginally the best for **P-2**.

These results indicate that, for problems with limited overnight flexibility, LDS and HLDS do not improve the performance of BT. This is intuitively correct, since the limited space of potential repairs to the current overnight distribution means that few sub-problems are setup for backtrack search, while the time-out does not allow large searches. However, the slightly better result for **P-2** hints that as the number of potential overnight distributions increases, the performance of the algorithms will improve further.

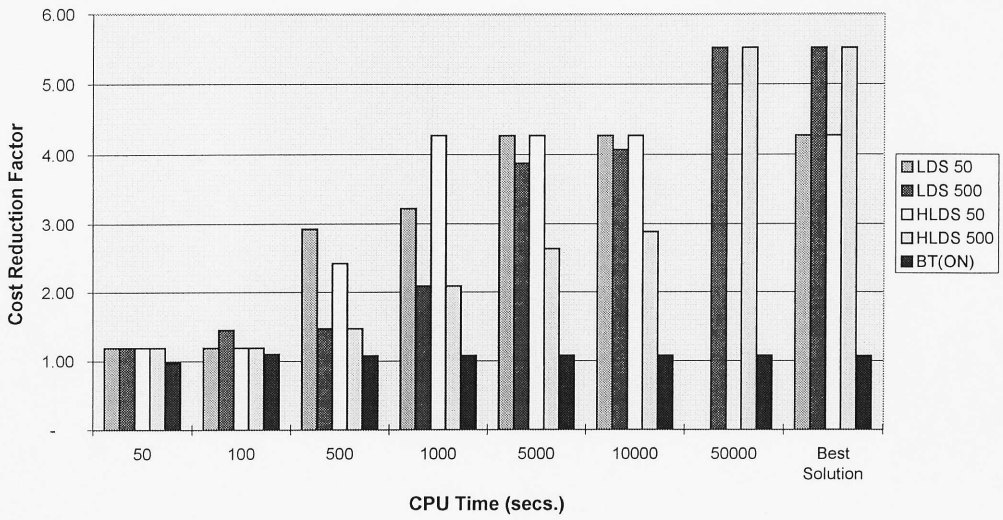


Figure 5.6: Optimization performance on **P-3**

The results for the larger two problems, **P-3** and **P-4**, confirm this very clearly. Performance dramatically improves on these much larger and more flexible problems, as Figures 5.6 & 5.7 show. Here five algorithms are compared with BT: LDS-50; LDS-500; HLDS-50; HLDS-500; and BT(ON).

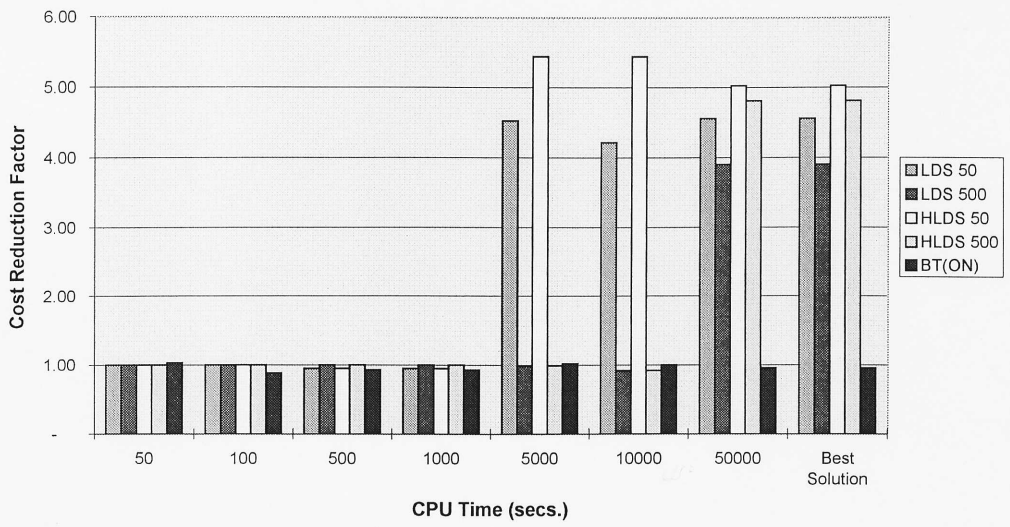


Figure 5.7: Optimization performance on **P-4**

On **P-3**, the performance of BT(ON) is very similar to BT, while the hybrid algorithms are characterized by dramatically increased optimization efficiency, achieving a CRF of 5.5 by the time the search ends. During the earlier stages of search, the biggest differentiator is the time-out. LDS-50 and HLDS-50 reach very good solutions at an early stage when compared with LDS-500 and HLDS-500, because they explore the search space more broadly. However, they terminate between 10,000 and 50,000 secs., again because they have exhaustively searched the repair space without finding better solutions within the backtrack time-out. At the final cut-off time, the 500 sec. algorithms have overtaken the 50 sec. algorithms. Another observation is that HLDS algorithms perform slightly differently from LDS.

The difference between LDS and HLDS is most pronounced on the largest benchmark, **P-4**. Among all four benchmarks, **P-4** involves the greatest distribution flexibility, with the greatest number of aircraft, fleets and stations. Again the LDS

algorithms are a substantial improvement on backtrack search and BT(ON). Also, the time-out remains the biggest differentiator among the LDS algorithms. However, HLDS consistently outperforms LDS. It is not known whether this improvement is because HLDS moves away faster from the airline's current assignment, or because it can re-visit sub-problems that have previously been searched, whenever new cost-bounds are obtained. The best performing algorithm overall on **P-4** is HLDS-50, due to the speed with which it generates good quality solutions, and its final CRF at the cut-off time.

A final point concerns the number of assignments in the solutions returned by the various algorithms that are different from the airline's current assignment. Table 5.3 shows that, for **P-1** and **P-2**, the best solutions obtained by the four tested algorithms are not especially practical, since they involve changing 31-45% of the airline's current assignment: the solutions that were generated earlier in the search would be more practical candidates for implementation. On **P-3** and **P-4** however, the best solutions obtained by the six compared algorithms are of greater practicality since they only change only 3-18% of the airline's operations. This is to be compared with the 65% change required by the "optimal" solution for **P-4**, as obtained by MIP.

Table 5.3: Percentages of flights changed in best solutions

Problem	MIP (Optimal)	BT	LDS-50	LDS-500	HLDS-50	HLDS-500
P-1	48%	45%	36%	—	36%	—
P-2	53%	31%	32%	—	32%	—
P-3	41%	5%	7%	9%	7%	9%
P-4	65%	3%	17%	14%	18%	17%

5.8 Related Work

A number of areas of research have contributed to the development of this hybridization technique.

Much related research has investigated the deficiencies of backtrack search on large-scale problems. Usually, this research advocates the use of local search instead. This is for two important reasons. The first is its *increased commitment flexibility.* In local search, poor quality assignments can be substituted at any point in the search. This is not the case in backtrack search where commitment is skewed (as discussed in Section 5.4.3). Choices made close to the root of the search tree are more difficult to change than those made nearer its leaves. The second advantage of local search is its *increased informedness.* Informedness is a characteristic of an algorithm that was empirically and probabilistically studied by Minton et al. on the one million queens problem [64], and is considered by them to be an important source of efficiency. They argued that local search makes better-quality search decisions, because, at any point in the search, a *combination* of search decisions is under evaluation. This combination of search decisions normally manifests itself as the *complete* assignment undergoing repair by local search. By contrast, the search-ordering heuristics of backtrack search are less informed, because they evaluate only smaller combinations of decisions: these are the *partial* assignments (the partial assignment consisting of the current labelling extended by an assignment to one of the unassigned variables).

If we extend Minton et al.'s argument, we are led to the conclusion that in unstructured problems such as N-queens, "informedness" is to some degree proportional to the length of the *partial* assignments under evaluation. This conclusion corroborates our experience on fleet assignment, even though fleet assignment is ostensibly a structured problem: an accurate picture of the true impact of a single variable assignment on feasibility and cost is much easier to obtain when most of the other plane type variables have been labelled. Clearly, the lengths of partial assignments assessed by backtrack search are especially small close to the root of the search tree, when few variables have been fixed. *Thus backtrack search seems to commit most strongly to its least "informed" decisions.* The hybridization technique avoids this pitfall by changing the commitment strategy close to the root of the search tree: it delegates early assignments to repair-based methods.

Another aim of the hybridization technique was to make early search choices that changed the character of the remaining problem, to make it more amenable to solution by backtracking. One way to change the problem's character is by *cutset decomposition*. Cutset decomposition transforms problems with a fully connected constraint graph into one with several disconnected constraint graphs. Conventionally, this is achieved within backtrack search, by labelling variables that are common to otherwise disconnected sub-problems [51, 16]. The transformed problems are easier to solve because they are more tightly constrained, and the sub-problems can be searched separately. Although the hybridization technique is not strictly a cut-set decomposition strategy (dependencies remain among the resulting sub-problems) it does use the repair-based strategy to transform the problem's characteristics for easier solution by subsequent backtrack search.

Also relevant is a strategy advocated by Tong el al., which involves a two-level algorithm hierarchy [86, 87]. The strategy was proposed for problems with "global constraints", i.e. ones that connect most of the variables of the problem. In their studies of floor-planning problems, the higher level algorithm heuristically assigned specific areas to rooms, such that the sum of their areas was equal to a total available area. The rooms' exact dimensions were then fixed at the lower level using backtrack search, such that the remaining constraints were satisfied. They noted that this was simply an instance of a more general hybridization strategy, where global constraints (such as the total area constraint) can be heuristically replaced by "low-level" constraints that guarantee to satisfy them. The lower level is then solved more easily using a backtrack search algorithm.

Clearly the hybridization technique is also related to Tong et al.'s suggested strategy. The repair-based method corresponds to their "high-level" procedure. The constraints on the overnight variables are constraints (i) and (ii), enforcing fleet sizes and maintenance requirements. These constraints do not individually satisfy Tong's definition of a global constraint, but collectively they make up what could

be described as a global constraint *network*. However, the two hybrid algorithms presented differ in two important respects. Firstly, the "high-level" repair-based methods implement a complete search that utilizes constraint propagation on all the problem constraints.[9] Secondly, they interleave both levels such that, at least in theory, complete algorithms may be constructed.

5.9 Conclusion

This case study investigated a hybridization technique designed to improve the performance of CP backtrack search on fleet assignment. Criticisms of the basic fleet assignment model, as well as the relative inflexibility of solution methods based on mathematical programming, encouraged us to find other, more effective methods within the CP paradigm. The experimental comparisons showed that the forward search hybrids described improve both the speed of the search, and the quality of solutions generated, for the largest of the commercial fleet assignment problems studied.

However, the fact remains that mathematical programming strategies are much more effective than ordinary CP approaches on the basic fleet assignment problem. To illustrate, when we ignore the requirement to remain close to the airline's current assignment, one MIP method takes less than 10 minutes to solve optimally our most difficult benchmark problem (problem **P-4**). The same problem could not be solved to optimality by any of the CP algorithms presented here, even after a whole day of execution. This level of performance suggests that such methods have much to offer CP algorithms if they are hybridized.

While this agenda is not pursued in the context of fleet assignment, the above conclusion motivated the work described next. In the third and final case study, we

[9] In Tong et al.'s approach, the "higher level" procedures are heuristic.

will demonstrate the potential of this type of hybridization, and examine why it is especially powerful.

CASE STUDY III

Chapter 6

Probe Backtracking

The hybrid algorithm described here was outlined first in [28], while the algorithm generalizations and the comparative study were presented in [29].

6.1 Introduction

This case study describes the application of the third hybridization form, **probe backtracking**, to **minimal perturbation scheduling**. The particular hybrid algorithm investigated, *unimodular probing*, interleaves backtrack search and linear programming. The linear programming solver acts as a forward search method, and is dynamically scoped to handle only a controlled subset of the problem constraints. The probe backtracking hybridization form applies a repair strategy to incorporate the values suggested by linear programming into the backtrack search.

The unimodular probing algorithm is designed to reconfigure schedules minimally, in response to a changing environment. External factors have caused an existing schedule to become invalid, perhaps due to the withdrawal of resources, or because

of changes to the set of scheduled activities. The total shift in the start and end times of already scheduled activities should be kept to a minimum. This optimization requirement may be captured using a linear optimization function over linear constraints. However, the disjunctive nature of the resource constraints impairs traditional mathematical programming approaches. *Unimodular probing* is compared with alternatives on a set of dynamic scheduling benchmarks, demonstrating its clear advantage over more conventional methods.

6.1.1 Overview

This case study considers a dynamic variant of the classic scheduling problem. The aim is to restore consistency in a schedule disturbed by resource or activity changes. To reduce disruption, the new schedule should keep changes to a minimum.

In the scheduling problems considered, the activities are not interruptible, and the resources are re-usable.[1] Dynamic variants of these NP-hard scheduling problems are theoretically and commercially significant. Our research into this subject was initially motivated by the need for an effective solution strategy to a commercial airline LSCO, aircraft utilization.

The kernel resource feasibility problem (KRFP), is introduced here to capture this scheduling problem and many others. The KRFP generalizes most scheduling benchmarks, including job-shop and resource constrained project scheduling. In the case study, minimal perturbation algorithms are compared on dynamic KRFPs.

The strategy proposed, *unimodular probing*, integrates backtrack search very closely with a forward search method, specifically a linear programming optimizer. It is a complete search algorithm that, at each search node, implements a repair step. The

[1] Re-usable resources, such as machines and containers, are available for re-allocation once activities using them terminate.

repair is to the latest assignment generated by the linear optimizer, which is always *super-optimal*: that is to say, it is better with respect to the optimization function than the *feasible* optimal, but only partially consistent. The sub-problem handled by the linear optimizer is dynamically scoped so that its solutions are always discrete (integer), facilitating their integration into CP backtrack search, and allowing discrete scheduling problems to be tackled efficiently.[2]

The strength of the strategy is demonstrated by comparisons on random benchmark tests. The tests are designed to capture the essence of the commercial scheduling application that motivated our research. The schedule disturbance inflicted in these tests (newly created resource contention) is the crux of dynamic scheduling problems.

The strategy's effectiveness raises the more general issue of perturbation minimization in constraint satisfaction problems, and suggests that the hybridization form utilized, *probe backtracking*, may be useful even when the cost function precludes the use of linear programming. In fact, probe backtracking could be used to support constraint satisfaction methods on problems without an optimization requirement, as remarked upon in the hybridization review (Chapter 7).

6.1.2 Outline of the Case Study

Section 6.2 starts with a toy example, and discusses the commercial importance of dynamic scheduling. The features of a minimal perturbation problem and a KRFP are then detailed in Section 6.3. The following section outlines a constraint model for the KRFP. A particular minimal perturbation variant of the KRFP, the resource utilization problem (RUP), will be used to illustrate and compare the algorithms throughout the case study. It is introduced in Section 6.5.

[2]Scheduling problems are often in reality discrete: for instance, the aircraft utilization problem that motivated this research required flights to be scheduled with a 5 minute resolution.

The second part of the case study describes and compares a range of algorithms. Section 6.6 studies the temporal sub-problem of the RUP and methods for solving it. The following section introduces a suite of algorithms for minimal perturbation. Unimodular probing is tested against more conventional hybrid algorithms, and against mixed integer programming (MIP) in Section 6.8. The final section discusses the reasons behind its good performance, and its wider applicability.

6.2 Motivation for Dynamic Scheduling

6.2.1 A Toy Example

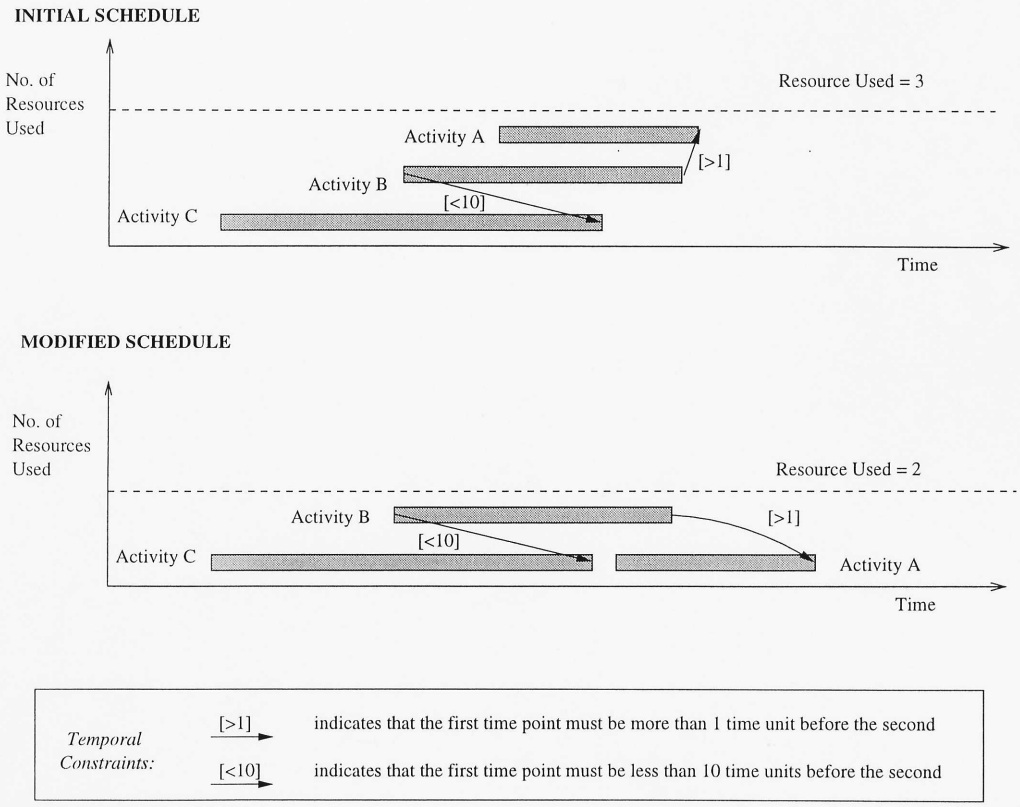


Figure 6.1: An example of a schedule modified to reduce resources

EXAMPLE : Figure 6.1 shows a toy dynamic scheduling problem. In the upper segment of the figure, three activities (A, B and C) are scheduled. The three utilize identical resources. Two temporal constraints, relating several of the start and end times of the activities, are satisfied in the current schedule.

The second segment of the figure shows the result of rearrangements in response to a reduction in the number of available resources. The modified schedule remains consistent with respect to the temporal constraints of the problem. □

This simple example shows the essence of dynamic scheduling. In this case study, the aim is to find a feasible, modified schedule that "minimally" differs from an old schedule that has become inconsistent.

6.2.2 Commercial Application

There is a strong commercial demand for techniques that manage better the process of schedule development. At the beginning of the scheduling process the schedule requirements are often vague, becoming well-defined only as the time of execution draws closer. Initial schedules are created only to be overtaken by newly emerging detail and unforeseen events. However, organizations exhibit inertia; changes must be agreed with affected parties, and operational factors obstruct schedule modifications. Disruption must be minimized as the schedule evolves.

Some research has focused on improving schedule **robustness** (see for instance [57] and [46]), a requirement that was noted earlier in Case Study II. The goal was to avoid changes to the working schedule by generating robust schedules (or scheduling decisions) that are likely to remain valid after minor changes to the problem definition. Nevertheless, major changes will necessitate modifications to even robust schedules. Another branch of research has investigated **dynamic scheduling**, where schedules are restored to feasibility when change is inevitable (e.g. reactive

scheduling in ISIS [31] and Micro-Boss [79]). [14] surveys research into robustness, dynamic scheduling and other related areas.

In this case study, the focus is on the second branch, dynamic scheduling. However, disruption is reduced by introducing an optimization criterion: in addition to restoring feasibility, the new schedule must differ minimally from the previous schedule.

Algorithms for dynamic scheduling deal with three forms of change resulting in schedule inconsistency. These are outlined below [with examples in brackets].

Activity changes. Requests for new or extended activities may lead to resource contention and schedule re-arrangements. In long-term scheduling, activities may be introduced to improve schedule efficiency, and make better use of under-utilized resources [temporarily leasing out cranes in construction scheduling]. In short term scheduling, new activities are introduced as they arise [just in time orders in out-sourced manufacturing], sometimes even during schedule execution [emergency services scheduling]. The durations of activities may be extended or their levels of resource usage increased for similar reasons.

Resource changes. Resource reductions also introduce schedule inconsistency. They can be requested over the long-term to reduce schedule costs [aircraft utilization problems]. In the short term, they are often in response to resource failures [out-of-service machines in job shop scheduling] that limit resource availability over all or part of the schedule horizon.

Temporal changes. The simplest form of temporal change is a contraction of the schedule horizon. Other temporal changes can also result in schedule inconsistency, and may occur in the long term [bus services re-arranged for regularity] or the short

term [downstream effects of missed activity deadlines in manufacturing].

While the unimodular probing strategy was originally developed to tackle resource reductions in aircraft utilization problems, the algorithms described handle all the above kinds of change, as detailed in Section 6.5.

6.3 Problem Definitions

6.3.1 The Minimal Perturbation Problem

The minimal perturbation problem is introduced to formalize the requirement for minimally disruptive rescheduling. Minimal perturbation problems correspond to the points where a solution (schedule) must change because the problem has been redefined. An old problem and a solution to it are given, as well as the required changes to the problem definition. Perturbation is modelled by a function that measures the difference between the old and new solutions. New solutions should minimize perturbation as measured by this function.

The scheduling problem is modelled here as a CSP, while the process of scheduling problem redefinition is captured as a *dynamic CSP* (DCSP). A DCSP is a sequence of CSPs, where each one differs from its predecessor by constraint additions or deletions [17]. Let us consider the case of a CSP Θ transformed into a new CSP Θ'. The transformation is captured in terms of two sets C_{del} and C_{add}, respectively standing for the set of constraints about to be removed from, and added to the constraints C of Θ (denoted C_{Θ}).

An optimal solution to a *minimal perturbation problem* is a solution to the altered CSP that is "minimally different" from a previously satisfactory solution.

Definition 6.3.1 ***Minimal Perturbation Problem***

A ***minimal perturbation problem*** Π *is a 5-tuple* $(\Theta, \alpha_\Theta, C_{\text{del}}, C_{\text{add}}, \delta)$ *where:*

- Θ *is a CSP;*
- α_Θ *is a solution to* Θ*;*
- C_{del}, C_{add} *are constraint removal and addition sets;*
- δ *is a function that evaluates the difference between two complete assignments. It is used to measure perturbation.*

A complete assignment is a ***solution*** *to* Π *if and only if it is a solution to* $\Theta' = (V, D, C_{\Theta'})$*, where* $C_{\Theta'} = (C_\Theta \setminus C_{\text{del}}) \cup C_{\text{add}}$*. A solution* α_Π *is* ***optimal*** *if and only if* $\delta(\alpha_\Pi, \alpha_\Theta)$ *is minimal.*

6.3.2 Introduction to the KRFP

The kernel resource feasibility problem (KRFP) is a generic model for scheduling. It constitutes the widest class of problems to which the algorithms presented may be applied without extension.

In the KRFP the goal is to fix the start and end times of activities such that quantities of available resource are not over-allocated. The decision variant of the KRFP is an NP-complete problem (it is an extension of the NP-complete resource constrained scheduling problem [35]). The KRFP allows the expression of variable duration activities and arbitrary metric constraints relating activity start and end times. The metric constraints can capture many of the complex temporal requirements that exist in real-world scheduling problems.

The KRFP is in fact the scheduling "kernel" of the resource feasibility problem

(RFP) of planning [24]. Unlike the RFP, the KRFP does not allow the representation of several resource pools of the same type, which means that every activity is committed to a single resource or resource pool. This restriction avoids the introduction of resource assignment sub-problems into the KRFP. Nevertheless, the algorithms described in this case study may be extended with resource assignment procedures to create algorithms for the full RFP.

Many well-studied scheduling problems, which were demonstrated in [23] to be RFP instances, happen also to be KRFP instances, including:

- the traditional scheduling benchmark, job shop scheduling [32]
- multiple capacity job shop scheduling [69]
- ship loading [2, 56]
- bridge building [4]

We shall describe the three components of the KRFP (activities, resources and time constraints) before giving a full problem definition. In the following, variable symbols are italicized (e.g. $quantity_i$) while constant symbols are not (e.g. r_i).

6.3.3 Activities in the KRFP

In the KRFP, the activities are represented as a set of n atomic activities $\mathcal{A} = \{A_1, ..A_n\}$, such that each activity:

1. requires only one type of resource;
2. is non-interruptible but possibly has a duration that may vary;
3. and uses a quantity of resource that does not change throughout its duration.

Each atomic activity $A_i \in \mathcal{A}$ consists of five elements:

- r_i, the name of the required resource type;
- $quantity_i$, the variable representing the used quantity of resource r_i;
- s_i, the *discrete* start point variable of A_i;
- e_i, the *discrete* end point variable of A_i;
- and $area_i$, the resource area variable of A_i.

A constraint $area_i = quantity_i \times (e_i - s_i)$ relates $area_i$ and $quantity_i$ to the duration of the activity. While both $area_i$ and $quantity_i$ are variables, in most problems at least one will be fixed. A fixed $quantity_i$ and a variable $area_i$ expresses a variable-duration activity such as the "block move" of blocks world planning. A fixed $area_i$ and a free $quantity_i$ represents an **elastic** activity, which trades-off duration against resource usage.

All other constraints relating the start and end times s_i and e_i should be included in the set of temporal constraints, $\mathcal{TC}$ (described in Section 6.3.5). For example, for an activity of fixed duration d, the single constraint $e_i = s_i + \mathrm{d}$ is added to $\mathcal{TC}$.

Complex activities that involve a sequence of sub-activities or use more than one resource must be modelled by several atomic activities. For instance, a complex activity using two resources $\mathrm{r}_1, \mathrm{r}_2$ must be represented as two separate atomic activities. Two temporal constraints equating their start and end variables must then be added to $\mathcal{TC}$, to ensure they coincide in time.

6.3.4 Resources in the KRFP

As mentioned, the KRFP resources are re-usable (they may be used again after activities terminate). They are described by:

- A set $\mathcal{R}$ of resource types.
- a function $quantity : \mathcal{R} \rightarrow \mathbb{N}$ specifying the number of interchangeable resources or the resource capacity for each resource type $r \in \mathcal{R}$

Let $\mathcal{A}_{\mathrm{r}}$ be the set of activities requiring a resource $\mathrm{r} \in \mathcal{R}$, given by $\mathcal{A}_{\mathrm{r}} = \{A_i : A_i \in \mathcal{A} \wedge \mathrm{r}_i = \mathrm{r}\}$. Let $\mathrm{T}_{\max}$ be the limit of the scheduling horizon. The following constraint ensures that the resource quantities are sufficient to satisfy the activities:

$$\forall r \in \mathcal{R}, \forall t \in \{0..\mathrm{T}_{\max}\} : quantity(r) \geq \sum_{A_i \in \mathcal{A}_r \wedge s_i \leq t < e_i} quantity_i$$

6.3.5 Temporal Constraints in the KRFP

$\mathcal{TC}$ is the set of all KRFP temporal constraints. $\mathcal{TC}$ consists of constraints in the following forms only:

- u R c (bounding constraints)
- u R v $\pm$ c (distance constraints)

where $\mathrm{R} \in \{ =, <, >, \leq, \geq \}, u, v \in \bigcup_{A_i \in \mathcal{A}} \{s_i, e_i\}$, and $\mathrm{c} \in \mathbb{N}$

Any set of temporal constraints expressed in these forms is tractable, since it corresponds to a simple temporal problem (STP) [19].

6.3.6 Definition of the KRFP

The full KRFP combines the activities, resources and temporal constraints. The decision problem variant of the KRFP is given below.

Definition 6.3.2 ***Kernel Resource Feasibility Problem*** (KRFP)
INSTANCE: *A 5-tuple* $(\mathcal{A}, \mathcal{R}, quantity, \mathcal{TC}, \mathrm{T}_{\max})$*:*

- *A set* $\mathcal{A}$ *of* n *activities* $A_1, \ldots, A_n$*. Each activity* A_i *is described as a 5-tuple* $(\mathrm{r}_i, area_i, quantity_i, s_i, e_i)$*.*
- *A set of* $\mathcal{R}$ *of* m *resource types* $\mathrm{r}_1..\mathrm{r}_m$*.*
- *The quantity of resource function* $quantity : \mathcal{R} \rightarrow \mathbb{N}$*.*
- *A set* $\mathcal{TC}$ *of temporal constraints of the form* u R c *or* u R $v \pm$ c*, where:* $\mathrm{R} \in \{ =, <, >, \leq, \geq \}$*,* $u, v \in \bigcup_{A_i \in \mathcal{A}} \{s_i, e_i\}$*, and* $\mathrm{c} \in \mathbb{N}$
- *The latest point of the scheduling horizon* $\mathrm{T}_{\max} \in \mathbb{N}$*.*

QUESTION: *Is there an assignment of values to the variables in* $\mathcal{A}$ *such that the following constraints are satisfied?*

- *The activity constraints :*

$$\forall A_i \in \mathcal{A}: \qquad area_i = quantity_i \times (e_i - s_i) \tag{6.1}$$

- *The temporal constraints in the set* $\mathcal{TC}$*, including the constraints relating the start and end times of each activity, as well as the constraints enforcing the schedule horizon* $\{0 \leq s_i \leq \mathrm{T}_{\max}, 0 \leq e_i \leq \mathrm{T}_{\max} : A_i \in \mathcal{A}\}$*.*

- *Let* $\forall r \in \mathcal{R}$: $\mathcal{A}_{\mathrm{r}} = \{A_i : A_i \in \mathcal{A} \wedge \mathrm{r}_i = \mathrm{r}\}$. *The resource constraints:*

$$\forall r \in \mathcal{R}, \forall t \in \{0..\mathrm{T}_{\max}\} : \qquad quantity(r) \geq \sum_{A_i \in \mathcal{A}_r \wedge s_i \leq t < e_i} quantity_i \tag{6.2}$$

6.4 Representing the KRFP as a CSP

This section gives the CSP used here to represent the KRFP. While the activity constraints of Eqn. 6.1 and the temporal constraints of $\mathcal{TC}$ can be represented directly, the resource overlap constraints of Eqn. 6.2 would impact efficiency, because a constraint would be required for every point in time.

The model presented here is due to [24]. A resource quantity variable Q_{r} is introduced for each $\mathrm{r} \in \mathcal{R}$. The resource quantity variable Q_{r} corresponds to the quantity used of that resource (the maximum quantity used over the schedule horizon). It is sufficient to count the quantities used at activity start times, since these are the points where increases in usage occur. For each activity $A_i \in \mathcal{A}_{\mathrm{r}}$, a variable $Q_{\mathrm{r}s_i}$ counts the quantity of resource r used at its start time s_i. Q_{r} is the maximum of these quantities $Q_{\mathrm{r}} = \mathrm{MAX}\{Q_{\mathrm{r}s_i} : A_i \in \mathcal{A}_{\mathrm{r}}\}$.

$Q_{\mathrm{r}s_i}$ variables are defined in terms of Booleans. A Boolean $B_{s_i A_j}$ is introduced for each pair of activities $A_i, A_j \in \mathcal{A}_{\mathrm{r}}$. It is set when activity A_j overlaps with s_i.

$$\forall r \in \mathcal{R}, \forall\ A_i, A_j \in \mathcal{A}_r : B_{s_i A_j} = \left\{ \begin{array}{l} 1 \text{ iff } s_j \leq s_i \wedge s_i < e_j \\ 0 \text{ otherwise} \end{array} \right\} \tag{6.3}$$

When linked with the corresponding $Q_{\mathrm{r}s_i}$, these Boolean variables link temporal and

resource reasoning. The following constraints are applied:

$$\forall r \in \mathcal{R}, \forall A_i \in \mathcal{A}_{\mathrm{r}} : \qquad Q_{rs_i} = \sum_{A_j \in \mathcal{A}_r} B_{s_i A_j} \cdot quantity_j \tag{6.4}$$

Each $Q_{\mathrm{r}s_i}$ variable on the LHS (left hand side) of one of the constraints of Eqn. 6.4 is bounded by the maximum resource quantity $quantity(\mathrm{r})$, via the constraints $Q_{\mathrm{r}s_i} \leq Q_{\mathrm{r}}$ and $Q_{\mathrm{r}} \leq quantity(\mathrm{r})$. While it is possible to dispense with Q_{r} and directly impose the constraints $Q_{\mathrm{r}s_i} \leq quantity(\mathrm{r})$, the Q_{r} variables will allow us to modify easily the KRFP in the next section, by adding new constraints on them.

6.5 A Minimal Perturbation Variant of the KRFP

This section introduces the resource utilization problem (RUP), chosen as an example of KRFP-based minimal perturbation problems. The algorithms of the case study are tested and compared on the RUP.

Section 6.2 described the kinds of changes that frequently arise in dynamic scheduling. These changes may cause the existing schedule to violate the following types of constraint:

- Type (1): the activity constraints of Eqn. 6.1
- Type (2): the temporal constraints in $\mathcal{TC}$
- Type (3): the resource constraints of Eqn. 6.2, modelled by Eqns. 6.3 & 6.4

All the algorithms presented will deal with change by solving the modified KRFP from scratch. The initial schedule impacts their search only via the minimal pertur-

bation requirement that is captured in the optimization function. *Thus any changes that result in a new KRFP can be handled.* Activity changes will cause the set of activities or the activity constraints to change; reductions in resource availability for all or part of the schedule horizon can result in changes to the resource quantities (all the horizon) or the introduction of new, dummy activities (part of it); and, lastly, any temporal changes cause modifications to the temporal constraint set.

Although the algorithms handle arbitrary KRFP changes, they address only a special class of perturbation functions, specifically those that can be expressed linearly (as a linear function over linear constraints). For example, a count of activities shifted in time cannot be expressed linearly, but a weighted sum of time shifts can. Nevertheless, the algorithms utilize search strategies that can be replicated in analogous algorithms tackling other perturbation functions. The wider applicability of probe backtracking in particular will be discussed later in Chapter 7.

Section 6.6.2 will show that, for the optimization functions addressed, schedules with violations of constraints of Type (1) and Type (2) can be cheaply transformed into minimally perturbed schedules with zero or more violations of Type (3) only. Those problems involving violations of Type (3) after the transformation are computationally difficult by comparison.

The RUP is of interest because it implements changes that always cause Type (3) violations directly. In the RUP, there is only one resource type, and activities require only one unit of that resource. The only allowed change to the problem is a reduction in the quantity of the resource. The RUP's perturbation function measures *temporal disruption*, defined to be the total time shift, i.e. the total change (usually in minutes) to the start and end times of activities. This is a linearly expressible function, as will be shown in the next section. The toy example shown in Fig. 6.1 involved an RUP-style change.

Definition 6.5.1 ***Resource Utilization Problem (RUP)***

A ***resource utilization problem*** *is defined to be a minimal perturbation problem* $(\Theta, \alpha_\Theta, C_{\text{del}}, C_{\text{add}}, \delta)$ *where:*

- Θ *is any KRFP* $(\mathcal{A}, \mathcal{R}, quantity, \mathcal{TC}, \mathrm{T}_{\max})$ *modelled as a CSP, such that:*
  - *there is only one resource type* r $(\mathcal{R} = \{\mathrm{r}\})$ *and* Q_r *is the resource quantity variable for that resource type*
  - $\forall A_i \in \mathcal{A} : quantity_i = 1$
- α_Θ *is a solution to* Θ
- $C_{\text{del}} = \emptyset, C_{\text{add}} = \{Q_\mathrm{r} \leq c\}$, *such that* $c < quantity(\mathrm{r})$
- $\delta(\alpha_{\Theta'}, \alpha_\Theta) = \sum_{u \in \bigcup_{A_i \in \mathcal{A}} \{s_i, e_i\}} |\alpha_{\Theta'}(u) - \alpha_\Theta(u)|$

The RUP is an interesting model in its own right. It may be used to capture minimal perturbation requirements in scheduling environments where resource reductions occur due to schedule efficiency improvements or resource failures. The following sections will show how the algorithms can solve minimal perturbation variants of the KRFP, but with particular emphasis on the RUP.

6.6 Properties of the Temporal Sub-Problem

With one exception (viz. mixed integer programming (MIP)), the algorithms compared in this case study are hybrids, conducting search in two, interleaved phases. In the resource feasibility phase, the algorithms relieve resource contention by ordering temporal variables. In the temporal optimization phase, the algorithms find values for the temporal variables that are optimal, *subject to the orderings decided in the*

resource phase. The two phases are interleaved differently in the algorithms, such that they all eventually find the problem's global optimum.

6.6.1 Minimal Perturbation in the Temporal Sub-problem

Let us focus initially on the temporal optimization phase. It was noted that the temporal constraints $\mathcal{TC}$ in a KRFP constitute a tractable sub-problem (Section 6.3.5). In this section, we sketch a proof that these linear temporal constraints can be combined to form linear constraint matrices that have a property known as *total unimodularity* (TU) [36].[3] Under ordinary conditions, the TU property holds for a linear constraint matrix *if and only if* the solution polyhedron defined by the linear constraints has discrete solutions at its vertices.[4] This is significant because many linear solvers such as Primal and Dual Simplex work by searching only those solutions at the vertices of the solution polyhedron.[5] Thus, for linear problems where the TU property holds, such solvers will always return optimal solutions that are *discrete* (integer) TU.

In the following, a set of constraints that always combine to form a TU matrix is said to ***have TU***, and a problem is said to have TU if its set of constraints does. We first prove that the temporal constraint set of the KRFP has TU. We then go on to prove that the TU property holds for an extended set of constraints, viz. one that additionally includes constraints linking the RUP perturbation function to the temporal variables.

TU problems are well known in the mathematical programming community. The constraints of ordinary network flow problems, such as *transportation*, *assignment*

[3]Linear programming (LP) solvers express the problem's variables and linear inequality constraints using a constraint matrix.

[4]These conditions are: (1) all constants are integer; and (2) the polyhedron is not empty [67].

[5]Simplex-based methods continue to be used widely, because of their average-case efficiency (refer to Section 2.4.3), despite being non-polynomial.

and *minimum cost network flow*, have TU. These are often encountered as components of combinatorial optimization problems.[6]

We relay below a practical set of conditions on the matrix that are sufficient to guarantee TU [45]. These conditions are satisfied by matrices expressing our temporal constraints.

A constraint matrix may be configured in two ways. In the first, rows represent constraints, and columns represent variables, with the element e_{ij} of the matrix denoting the coefficient of a variable j in a particular constraint i. In the dual configuration, the reverse holds, with rows corresponding to variables and columns to constraints.

A set of sufficient conditions in a problem or its dual:

1. All variable coefficients are 0, 1, or -1, and all constants integer.
2. Two or less nonzero coefficients appear in each row.
3. The columns of the matrix can be partitioned into two subsets S_1 and S_2 such that
   (a) If a row contains two nonzero coefficients with the same sign, one element is in each of the subsets.
   (b) If a row contains two nonzero elements of opposite sign, both elements are in the same subset.

[6]For example, the basic fleet assignment problem described in Chapter 5 is a *multi-commodity network flow* problem. While the set of problem constraints do not have TU, the underlying network flow sub-problem means that optimal solutions to the problem's linear relaxation often come close to integrality, making LP-based approaches efficient.

Lemma 1: *The set of KRFP temporal constraints has TU.*

Proof:

All the unary (bounding) temporal constraints in the matrix rows have only one nonzero coefficient. All the binary (distance) temporal constraints involve two nonzero coefficients of opposite sign. The variables (matrix columns) can be partitioned into a subset S_1 containing all columns, and another S_2 containing none. Thus the temporal constraints of the KRFP (and thus the RUP) satisfy the sufficient conditions as outlined above.

Often a sub-problem satisfies the conditions guaranteeing the TU property but *link constraints* violate them. Link constraints are not genuine problem constraints because they do not constrain the set of feasible solutions. They are used to link the variables of the sub-problem having TU with variables in the optimization function. The RUP optimization function uses link constraints.

The RUP perturbation function δ was defined to be the sum, over the temporal variables, of absolute change. This function is modelled by introducing a variable d_x for each temporal variable x. d_x corresponds to the absolute change in x and is given by $d_x = |x - \mathrm{c}|$, where c represents the value given to x in the previous solution. This non-linear expression is expressed linearly by placing d_x in the optimization function and adding the following linear constraints:

$$d_x \geq x - \mathrm{c} \tag{6.5}$$

$$d_x \geq \mathrm{c} - x \tag{6.6}$$

When the constraints of Eqns. 6.5 & 6.6 are included in the constraint matrix, it ceases to satisfy the sufficient conditions for TU. However, it is proved below that

the property of TU is not lost when they are added to a set of constraints having TU.

Lemma 2: *If E is a set of constraints having TU then $E \cup \{d_x \geq x - \mathrm{c}, d_x \geq \mathrm{c} - x\}$ has TU, if d_x is unconstrained in E.*

Proof:

The proof shows that the new vertices created by the addition of the two constraints are also discrete.

Let E be a set having TU. Let E be extended to E' by adding the new variable d_x and the constraint pair of Eqns. 6.5 & 6.6. d_x adds a new dimension to the polyhedron, and cuts it along the hyper-planes $d_x = x - \mathrm{c}$ and $d_x = \mathrm{c} - x$. All existing vertices are projected in the d_x dimension onto these two hyper-planes. At the projected vertices d_x takes the value of either $x - \mathrm{c}$ or $\mathrm{c} - x$. Both values are discrete because c is an integer constant and x is discrete at the projected vertex since E has TU. In addition, new vertices are added where the two new hyper-planes intersect at $x = \mathrm{c}$. But the hyper-plane $x = \mathrm{c}$ corresponds to a unary constraint with an integer constant, which could be added to E without losing TU. Thus the new vertices are also discrete, and thus the extended set E' also has TU.

Theorem: *The union of the set of KRFP temporal constraints, and the set of RUP perturbation function constraints, has TU.*

Proof:

By induction over constraint pair addition. Base Case: The set of KRFP temporal constraints has TU (Lemma 1). Inductive Step: If E is a set of constraints having TU then $E \cup \{d_x \geq x - \mathrm{c}, d_x \geq \mathrm{c} - x\}$ has TU, where d_x is unconstrained in E (Lemma 2).

6.6.2 Transforming All Schedule Violations into Resource Constraint Violations

The proof shows that the entire temporal optimization phase can be solved efficiently, since cheap linear solving procedures will guarantee to return globally optimal, discrete solutions.

We affirm now the statement in Section 6.5 that schedules with violations of Type (1) (activity constraints) and Type (2) (temporal constraints) may be transformed efficiently into violations of Type (3) (resource constraints), such that the newly transformed schedule is not more perturbed than any feasible schedule. This new schedule is described as being *super-optimal*; a super-optimal assignment is a partially consistent assignment that is at least as good, with respect to the optimization function, as any solution (i.e. fully consistent assignment).

The simple case, Type (2) violation, is addressed first. It was shown that the temporal and optimization function constraints constitute a TU set. Because of the TU property, a linear solver will produce a new schedule satisfying the temporal constraints while minimizing changes to the existing (temporally inconsistent) schedule. The new schedule however may still violate constraint Types (1) and (3).

To transform violations of the activity constraints – Type (1) – additional steps are required before and after the linear solver is invoked. The activity constraints $area_i = quantity_i \times (e_i - s_i)$ have four variables only two of which appear in the optimization function, namely e_i and s_i. These two variables must be minimally constrained so that no optimal solutions are ruled out. This is achieved by the imposition of two constraints $LB(area_i/quantity_i) \leq (e_i - s_i)$ and $(e_i - s_i) \leq UB(area_i/quantity_i)$, where LB an UB are functions returning respectively the integer lower and upper bounds of an expression. These two constraints can be expressed in the forms required for inclusion in $\mathcal{TC}$, the temporal constraint set. The

linear solver then returns a discrete optimum satisfying $\mathcal{TC}$ and suggesting specific values for s_i & e_i. The final step is to re-satisfy the activity constraints using a polynomial-time procedure that chooses appropriate values for $area_i$ and $quantity_i$. Any remaining violations in the super-optimal schedule will now be of Type (3).

6.7 A Suite of Algorithms for Minimal Temporal Perturbation

Let us turn our attention now to the resource feasibility phase of the algorithms. Given the efficient algorithms for temporal optimization these are the critical algorithmic component.

Unlike a traditional backtrack search procedure using conventional variable and value ordering heuristics, the algorithms for the resource feasibility phase configure their search around the problem's resource constraints (Eqn. 6.4). Search progresses by reducing *contention* in these constraints. We give here a utilitarian definition for contention: *the level of contention in a constraint is the ratio of the number of nodes in the search space where the constraint is violated, to the total number of search nodes in the search space.* Note that since the search space is dependent on the actual algorithm applied, constraint contention is also dependent on it (Section 2.2).

For some algorithm and constraint combinations, contention can be estimated for an individual constraint, and the search can be directed towards search regions where its levels of contention are reduced. The algorithms for the resource feasibility phase reduce contention in a resource constraint by imposing new temporal orderings, using a backtrack search procedure. Once the resource feasibility stage ends, the imposed temporal orderings will ensure that the temporal optimization phase does not violate any of the resource constraints.

Recall that, in the RUP, only one resource type r exists, and each variable Q_{rs_i} corresponds to the quantity of that resource used at the start of activity A_i (Section 6.4). Since each RUP activity uses only a single unit of the resource, the resource constraint of Eqn. 6.4 simplifies to:

$$\forall\ A_i \in \mathcal{A}: \qquad Q_{rs_i} = \sum_{A_j \in \mathcal{A}} B_{s_i A_j} \tag{6.7}$$

Assume that no local consistency checking is applied to the above constraint. The left hand side (LHS) of the equation, Q_{rs_i}, is otherwise constrained only by the RUP's *reduced* resource capacity: c (c < *quantity*(r) via $Q_{rs_i} \leq Q_r$ and the newly added constraint $Q_r \leq$ c (Def. 6.5.1). On the RHS, the sum expression will depend on the domains of the overlap Booleans, which in turn depend on the domains and orderings of variables in the temporal sub-problem. Notice that during search only the RHS of the equality will change, since the resource feasibility phase conducts search through the addition of temporal orderings. The constraint is only violated when the RHS exceeds the LHS.

Thus contention exists in the resource constraints of Eqn. 6.7 only when the RHS upper bound exceeds the LHS upper bound, since there exist search nodes where violation could occur. Moreover, the degree of contention is proportional to the difference between the two bounds, and can be estimated by $max(0, UB(RHS) - UB(LHS))$, where UB is a function returning the upper bound of an expression according to the domains of the variables in it. By continually monitoring the value of this expression for the resource constraints during search, the search procedure may reduce contention in the constraint with the highest estimated contention. Once a constraint is selected for contention relief, a backtrack-able search decision imposes a new temporal constraint that forces apart two temporal variables, setting some RHS Boolean to 0. This decision is backtracked later if inconsistency is detected,

or if a solution has been found but a better one is sought by the branch and bound optimization process.

We turn our attention back now from the RUP to the general KRFP. Here, the resource constraints of Eqn. 6.4 do not necessarily simplify to Eqn. 6.7, and a minor extension is needed to allow search to relieve contention using an alternative method. Recall that elastic activities have variable resource usage. Where variable resource usage occurs on the RHS, we can also lower $UB(RHS)$ by reducing the upper bound of a quantity variable $quantity_j$. This is reduction is obtained by forcing apart the temporal variables s_j, e_j of some elastic activity A_j (to impose a greater minimum duration and consequently a reduction in $UB(quantity_j)$).

6.7.1 The Constraint Backtrack Algorithms (CB)

In the first set of algorithms, resource feasibility phase algorithm is designated constraint_backtrack (CB). This is a generalization of a resource feasibility algorithm described in [71].

Fig. 6.2 gives the pseudo-code for constraint_backtrack. The procedure is initialized with the parameter MonitoredConstrs set to be the constraints where contention must be relieved (namely the resource constraints). The remaining arguments OtherVarDoms and Constrs are set respectively to the problem's variable domain pairs, and to its full constraint set. Calls to constraint_backtrack will return either $\perp$ indicating that the problem is infeasible, or new values for OtherVarDoms and Constrs otherwise. In the latter case, the values include the new temporal ordering constraints imposed to guarantee resource feasibility, as well as their inferred consequences (viz. associated reductions of the variable domains). These new values are the input to the temporal optimization phase.

The first step in the constraint_backtrack procedure is to apply local consistency

checking (represented as a call to consistency_check(OtherVarDoms, Constrs)). The type of consistency checking applied is considered later. For the time being though, we note that this call returns either $\perp$, indicating inconsistency has been detected, or alternatively a new OtherVarDoms and Constrs pair, updated with the results of the consistency checking.

```
1.  begin constraint_backtrack(MonitoredConstrs, OtherVarDoms, Constrs)
2.      Result := consistency_check(OtherVarDoms, Constrs)
3.      if Result == ⊥
4.          then return ⊥
5.      (OtherVarDoms',Constrs'):= Result
6.      ContentionConstrs := constraint_filter(MonitoredConstrs)
7.      if ContentionConstrs == ∅
8.          then return (OtherVarDoms',Constrs')
9.      Constr := select_constraint(ContentionConstrs)
10.     DecisionConstr:= select_decision(Constr)
11.     Constrs'' := Constrs'∪{DecisionConstr}
12.     Result:= constraint_backtrack(MonitoredConstrs, OtherVarDoms',Constrs'')
13.     if Result ≠⊥
14.         then return Result
15.     Constrs'':= Constrs' ∪ {not(DecisionConstr})
16.     return constraint_backtrack(MonitoredConstrs, OtherVarDoms', Constrs'')
17. end constraint_backtrack
```

Figure 6.2: constraint_backtrack (CB): A procedure for the resource feasibility phase

Constraint filtration and selection

The next step is the call to constraint_filter, which returns the constraints that are subject to contention. The constraints filtered out have no impact on resource feasibility in this search branch, so they are excluded from consideration. When no more contention constraints exist, CB can terminate, passing control over to the temporal optimization phase (subsequent backtracking may return search again to CB, if better solutions are sought by the branch and bound process).

The procedure select_constraint chooses from the filtered constraint set (Contention-

Constraints) the constraint Constr with the greatest potential for conflict – viz. the greatest value for the RHS upper bound minus the LHS upper bound. This value is always positive for ContentionConstraints.

Selecting a decision

At this point the objective is to reduce contention in Constr. Suppose that Constr constrains the resource quantity Q_{rs_i} at the start of activity A_i. As explained, in the RUP contention is reduced by imposing a new temporal ordering constraint, setting some RHS $B_{s_iA_j} = 0$. In the CB algorithm, the new temporal ordering constraint is chosen using a heuristic designed to select the one with the smallest impact, in order to maximize the chance of obtaining feasible solutions. This is an example of the succeed first principle (SFP) introduced in Section 2.3.4. The heuristic in select_decision assesses, for each RHS $B_{s_iA_j}$ that is not yet fixed, the ordering constraints that could be used to set $B_{s_iA_j} = 0$ (namely $s_i < s_j$ and $e_j \leq s_i$). If a potential ordering constraint is compatible with the variable domains, the local impact of imposing the constraint is calculated, in terms of the number of values pruned from the arc-consistent domains of s_i, s_j and e_j. DecisionConstr is assigned to the ordering constraint with the smallest local impact.

In the recursive call to constraint_backtrack, DecisionConstr is added to the set of constraints to produce Constr''. On backtracking DecisionConstr is replaced by its negation not(DecisionConstr) (as expected not($x < y$) is simply $x \geq y$ and not($x \leq y$) is $x > y$).

Constraint propagation

We now return our attention to the procedure consistency_check. On the recursive call to constraint_backtrack, consistency_check establishes arc-consistency in the prob-

lem variables.[7] The latest decision made is propagated through interplay between these resource and temporal constraints. Notice that propagating the decision may result in confirmed overlaps elsewhere (e.g. $B_{s_i A_k} = 1$ for some k), changing the levels of contention in the resource constraints.

In addition to arc-consistency a specialized look-ahead check is applied. This check is found to be important for the KRFP model, which constrains overlap at activity start times only. Whenever a Boolean $B_{s_i A_j}$ is set to 0 by a search decision, a resource usage profile is built for the time interval that activity A_i must span (from $UB(s_i)$ to $LB(e_i)$). This interval exists only when $UB(s_i) < LB(e_i)$. Other activities are assumed to take minimal spans over this interval. Failure is signaled and a backtrack instigated when resource usage exceeds the capacity at any point in the interval.

Increasing propagation in the algorithm: (CB+LP) and (CB+LP+Edge)

As the results of Section 6.8 will show, the CB hybrid is relatively ineffective at obtaining the optimum and proof of optimality. The propagation methods described above fail to prune search sufficiently. In particular, the optimization function is not used to reduce the search space during the resource feasibility phase; the implications of individual search decisions for cost are not discovered until the temporal optimization phase, when the temporal variables are fixed.

Cost bounds can dramatically reduce the search space of the optimization process by pruning search branches that lead to worse solutions than those already found. The algorithm CB was extended to a new algorithm CB+LP applying the linear solver during the resource feasibility phase. The linear relaxation of the problem is solved after every search decision to obtain a lower bound on the cost in the resulting sub-problem. This propagation is referred to as *LP cost bound propagation.*

[7] Specifically it applies arc-B consistency on the arithmetic constraints [58], and specialized AC-5 procedures for the logical constraints [21].

The algorithm was also extended to deploy the cubic edge-finder for capacitated resources [68]. This algorithm is denoted CB+LP+Edge and also achieves improvements in performance. Edge is more powerful but more costly than the look-ahead check described earlier.

Nevertheless, even the improved search spaces of CB+LP and CB+LP+Edge remain large. For improved performance, it is necessary to reduce the number of decisions made in the resource feasibility phase, avoiding, wherever possible, modifications to the old solution.

6.7.2 The Probe Backtrack Algorithms (PB)

Probe backtracking is an extension of constraint backtracking, where backtrack search is supported by a **forward search** procedure (*a probe generator*) that generates good quality, but generally only partially consistent assignments (*probes*).

The origins of the generic probe backtracking algorithm are described later (Section 6.9.2). Probe backtracking associates each unassigned variable not only with a domain of values, but also with a tentative value. In probe backtracking the tentative value is not necessarily adhered to and may be changed by the probe generator as the backtrack search progresses. The purpose of the probe generator is to direct the backtrack search component and limit the size of its search space: the backtrack search is focused only on regions where the probe violates constraints. Probe backtracking is another way of creating complete, repair-based algorithms.[8]

Figure 6.3 demonstrates the extension of the CB algorithm to perform a probe driven search. The changes have been highlighted in the figure. The first change is the introduction of obtain_probe_assignment near the start of the procedure. This represents a call to the probe generator, which obtains the probe used to focus the

[8]Other examples include the LDS and HLDS algorithms of Case Study II.

search decisions at this search node. The only condition that must be guaranteed in a probe generator is that the assignments it produces must lie within the variables' up-to-date domains. In practice the other condition necessary for good algorithm behavior is that the probe generator produces good quality assignments. In optimization problems, probes should ideally be super-optimal.

```
1.  begin probe_backtrack(MonitoredConstrs, OtherVarDoms, Constrs)
2.      Result := consistency_check(OtherVarDoms, Constrs)
3.      if Result == ⊥
4.          then return ⊥
5.      (OtherVarDoms',Constrs'):= Result
6.      ProbeAssignment := obtain_probe_assignment(OtherVarDoms', Constrs')
7.      ViolatedContentionConstrs := constraint_filter(MonitoredConstrs, ProbeAssignment)
8.      if ViolatedContentionConstrs == ∅
9.          then return (OtherVarDoms',Constrs')
10.     Constr := select_constraint(ViolatedContentionConstrs, ProbeAssignment)
11.     DecisionConstr:= select_decision(Constr, ProbeAssignment)
12.     Constrs'':= Constrs'∪{DecisionConstr}
13.     Result:= probe_backtrack(MonitoredConstrs, OtherVarDoms',Constrs'')
14.     if Result ≠⊥
15.         then return Result
16.         else
17.             Constrs'':= Constrs' ∪ {not(DecisionConstr})
18.             return probe_backtrack(MonitoredConstrs, OtherVarDoms', Constrs'')
19. end probe_backtrack
```

Figure 6.3: probe_backtrack (PB): An improved, probe backtracking algorithm for the resource feasibility phase

The other highlighted changes to the algorithm reflect the fact that the assignments returned by the probe generator are now inputs to the constraint_filter, select_constraint and select_decision procedures. These are re-examined below.

Constraint filtration and selection

It is now possible to increase the number of constraints removed by the filtration process. As with the CB algorithms, any constraints that are not subject to con-

tention are filtered out. In addition, if the tentative values of the probe returned by the probe generator satisfy a constraint, then it is filtered out, even though it may still be subject to contention.

EXAMPLE : Let $X[D\mathrm{v}]$ represent a variable X with a domain D and a probe value v. In probe backtracking, ViolatedContentionConstrs would not include the following constraint, because it is satisfied in the probe assignment.

$$Q_{rs_i}[\{0..1\}1] = B_{s_iA_j}[\{0,1\}0] + B_{s_iA_k} : [\{0,1\}0] + B_{s_iA_l}[\{0,1\}1]$$

This is in spite of the fact that it is subject to contention. □

While constraints satisfied by the current probe may at some later point become violated in a future probe, for now it is assumed that the current probe will lead to a solution. Clearly, the procedure may terminate immediately if all constraints are satisfied by the current probe, since it is a solution.

Selecting a decision

In decision selection the assignment is also used to order the possible choices. Since the constraint Constr is violated by the probe assignment, decisions should be selected that ensure new probes obtain assignments that are closer to satisfying Constr. Selecting decisions that allow the probe generator to return the same probe assignment would not progress the search.

In the RUP, Constr is a constraint on the quantity of used resource Q_{rs_i} at the start of activity A_i. As for CB, contention reduction is achieved by forcing apart two temporal variables (i.e. imposing a new temporal ordering constraint) setting some RHS $B_{s_iA_j} = 0$ in the process. However, in PB the heuristic prefers to reset a $B_{s_iA_j}$ that has a value of 1 under the current probe assignment. In other words,

only those temporal ordering constraints that would remove an overlap in the probe are considered as candidates for DecisionConstr. The heuristic operates like a filter because such overlaps always exist in probes violating the constraint.

Subject to this primary heuristic, a secondary heuristic is used to order the decision constraints. Two secondary heuristics were considered. The first is the SFP heuristic used for decision selection in the CB algorithm (Section 6.7.1). Unless indicated otherwise, this is the secondary heuristic applied in the PB algorithms studied.

The other, alternative secondary heuristic estimates the impact of the new ordering constraint on the probe assignment, in terms of the amount of temporal shift its variables must undergo. The following example serves to illustrate how this impact is estimated.

EXAMPLE : Suppose that the temporal ordering constraint $e_j[D_{e_j} \mathrm{v}_{e_j}] \leq s_i[D_{s_i} \mathrm{v}_{s_i}]$ is being evaluated as a candidate DecisionConstr.[9] Being a candidate, the constraint must currently be violated ($\mathrm{v}_{e_j} > \mathrm{v}_{s_i}$ in the probe assignment), otherwise it would have been filtered out by the primary heuristic. The alternative secondary heuristic calculates a lower bound on the amount of change (temporal shift) that must be undergone by the probe, in the event the constraint is applied. The chosen candidate minimizes this value. Here, the lower bound is given by $\mathrm{v}_{e_j} - \mathrm{v}_{s_i}$. □

The alternative secondary heuristic selects as the DecisionConstr the ordering constraint with the smallest impact on the probe. In the experimental section, algorithms deploying the alternative secondary heuristic will be suffixed by +Alt (e.g. PB(Unimod)+Edge+Alt).

We now make a more general point on decision making in probe backtrack algorithms. As in the CB procedure, the PB procedure of Fig. 6.3 searches a binary

[9] As before, $X[D\mathrm{v}]$ represents the variable X together with its domain D and probe value v.

tree (at each search node, the algorithm partitions the sub-tree into two smaller branches, corresponding to the assertion of DecisionConstr or its negation). In general, however, the probe backtrack algorithm can make decisions at a search node that correspond to more than two branches (for example, a simple alternative could be to exhaustively label the variables of a violated constraint). However, for efficiency and completeness, the decisions made at each search node must always partition the sub-tree fully.

The algorithm PB(Old)

The first version of a probe backtracking algorithm, PB(Old), creates a probe generator closely based on the old solution. The old solution is of course only partially consistent with the new problem constraints. Moreover, during search, variable domains are pruned further by the local consistency propagation of any search decisions. In PB(Old), the probe values for the pruned temporal variables are moved minimally to lie again within their domains whenever obtain_probe_assignment is invoked. The values for the Boolean variables are then made to correspond with the values chosen for the temporal variables.

Although this probe is super-optimal, it has one serious flaw: the probe values may not satisfy the binary constraints of the temporal sub-problem. Since these constraints are not monitored and fixed (the MonitoredConstrs parameter includes only resource constraints), their violations will remain undetected until the temporal optimization phase. At that point, a different schedule might be generated by the temporal optimizer, which guarantees to satisfy all temporal constraints. However, this different, temporally feasible schedule may now introduce new resource constraint violations, causing the algorithm to output infeasible solutions.

The algorithm is restored to soundness by introducing an intermediate phase between the probe backtracking and temporal optimization phases. The aim of the

intermediate phase is to relieve any remaining contention from resource constraints that were filtered out, because they did not violate the last probe. The backtrack search of the intermediate phase posts new ordering constraints that do not contradict the orderings implicit in the probe, unless local consistency checking identifies an inconsistency in the temporal constraints, and causes a backtrack.

Used in conjunction with LP propagation on the relaxed problem, the resulting algorithm, PB(Old)+LP, will be shown to perform better than the CB algorithms. This is because it focuses search to a degree on the old schedule, preferring temporal orderings that change it less radically than those of the CB algorithms.

Nevertheless, the algorithm is not ideal; the probes generated in the probe backtracking phase are of gradually decreasing relevance as search progresses. This is because they will violate a number of temporal constraints that increases as new search decisions are made. Though the tentative values for the remaining variables will remain super-optimal, they may violate so many temporal constraints that they no longer provide accurate information about the optimum solutions to the temporal sub-problem.

The Unimodular Probing algorithm PB(Unimod)

This observation motivates a probe backtracking algorithm that continually updates the probe assignment during search to satisfy the temporal constraints. The global impact on cost of choices made during search are clarified by finding the LP optimum given the temporal constraints posted so far. Thus LP is dynamically scoped to handle a changing set of temporal constraints — the linear relaxations of any other constraints are *not* included in the LP set, to ensure discrete probes. The discrete values at the LP optimum are installed as the new probe values for the variables. Since only violations of resource constraints occur in this probe, it achieves a high level of consistency, and therefore is a better approximation of the problem's optimal

solutions. The probe backtracking algorithm described is a *unimodular probing* algorithm [27, 28, 29].

At each search step the unimodular probing algorithm applies LP to the up to date TU set $\mathcal{TC}'$. The discrete, super-optimal assignments returned by LP, called unimodular probes, satisfy the constraints in the TU set. As with PB(Old), PB(Unimod) focuses on the resource constraints that are violated by the probe. If the probe violates any resource constraints, the algorithm selects a violated constraint, and rules out the present unimodular probe by imposing a new temporal ordering constraint. Thus search decisions add at least one new TU-preserving constraint to the LP set at every search node. After the decision is posted, other unary and binary TU constraints will often be derived by the local consistency methods. Any such constraints are also included in $\mathcal{TC}'$.

When no more resource constraint violations remain, the PB(Unimod) probe will be a full solution, unlike the probe of PB(Old). The solution will also be optimal subject to the temporal orderings imposed so far, and no further search needs to take place in this search branch. Backtracking can therefore be immediately initiated in a branch and bound search for the global optimal.

To summarize, PB(Unimod) extends backtrack search by using LP not only as a propagation method (detecting failure through cost bounds), but also as the basis for variable and value selection heuristics in discrete problems such as the KRFP. Most importantly, the repair strategy reduces the number of decisions made by the backtrack search component. This is by avoiding further decisions in a search branch where the conflict set has become empty. An empty conflict set is proof that the probe is both feasible *and optimal given the search decisions taken.* Local consistency techniques such as arc-consistency and edge finding are orthogonal and contribute to the efficient handling of the disjunctive constraints outside LP's scope.

6.8 Performance Comparisons

Unimodular probing was found to outperform significantly its rival algorithms on the LSCOs that originally motivated this research. However, here the algorithms are tested systematically, on randomly-generated RUP benchmarks. The commercial LSCOs differed in that they involved a greater number of activities, and had more structured temporal subproblems than the benchmarks. Nevertheless, the benchmarks appear to be closely related to the commercial problems investigated.[10] Details of the benchmarks have been made available on the Internet for future comparisons.[11]

Each benchmark is generated by randomly creating a set of activities for a single resource pool over a restricted schedule horizon. Feasible binary temporal constraints are posted such that the expected number of constraints on a temporal variable is given by E_{density}, where E_{density} is a parameter to the problem generator. While activities have a nominal duration, this is not fixed: activities are allowed to shrink by a given amount.

For every benchmark file a number of different problems may be constructed. One parameter is the maximum allowed time shift, which imposes lower and upper bounds on the temporal variables. In the problems tested bounds on acceptable time shifts are uniformly applied to all the temporal variables (e.g. activity starts and ends were allowed to shift by no more than ± 50 minutes). The second parameter is the desired reduction in the number of resources. Greater reductions tighten the problem and cause greater perturbation.

Three problem sets were used in the comparison. In all three, the activity durations were nominally 100 minutes and the maximum allowed shrinkage 10. The

[10] In fact, a deeper understanding of algorithm performance issues gained from this empirical study has enabled a number of improvements to the commercial algorithms.

[11] http://www.icparc.ic.ac.uk/ ~hhe/rfp_benchmarks.html

total schedule horizon was 985 minutes with a time granularity of 5 minutes. All algorithms were tested on a Dell Optiplex GXa (Intel Pentium II at 300Mhz), with a timeout of 2500 CPU seconds and a spaceout of 150MB. The algorithms were implemented in the ECL^{i}PS^e CLP platform, and both PB algorithms utilize the ECL^{i}PS^e Repair library [26].[12]

In problem set 1, the following parameters were varied: no. of activities ({20,30}); E_{density} ({0,0.5,1}); maximum time shift ({±20,±50,±100}); and number of resource reductions ({-1,-2}). 50 problem instances were tested at each point, to create a total of 1800 problems.

Problem set 1 was searched by five algorithms. Four of the algorithms compared on this problem set have been described in this case study: CB,CB+LP,PB(Old)+LP, PB(Unimod). The fifth, MIP, was a commercial mixed integer programming package: CPLEX. The same problem formulation was used, with the problem Booleans declared as integer. It should be noted that there exists a large body of literature on specialized MIP problem formulations that can take advantage of specific problem characteristics (e.g. [74]). However we are not aware of any formulations that closely match the RUP or the KRFP more generally. The CPLEX automatic configuration settings were used; other settings were attempted on a representative sample of 200 problems for both the node selection and variable selection strategies with minor or no improvement (3% in the best case). MIP practitioners confirm our understanding that CPLEX's automatic configuration settings are usually among the best.

The results are shown in Table 6.1. The first column gives the percentage of problems where a timeout or spaceout occurred. The second gives the percentage for which the algorithm successfully proved a solution to be optimal. The final column gives the percentage of problems proved to be infeasible.

[12]Problem sets 1 and 2 are both based on the schedule instances 1-50 on the web-site, while problem set 3 is based on the schedule instances 1-100.

Table 6.1: Problem set 1: Search results

Algorithm	Time/Spaceouts	Proofs(Optimal)	Proofs(Infeasible)
CB	50.8%	26.3%	22.8%
CB+LP	27.3%	50.4%	22.2%
PB(Old)+LP	22.2%	53.1%	24.7%
PB(Unimod)	**7.8%**	**66.2%**	**26.0%**
MIP	24.0%	57.7%	18.3%

The results show an improvement that corresponds with the progression of algorithms from CB (the worst) to PB(Unimod) (the best). MIP appears to hold out relatively well, albeit more so on proving optimality than on proving problem infeasibility. Figure 6.4 shows the percentages of time/spaceouts for the algorithms distributed over problem size (20/30 activities). The graph indicates that the computational advantage of PB(Unimod) scales as the number of activities increases.

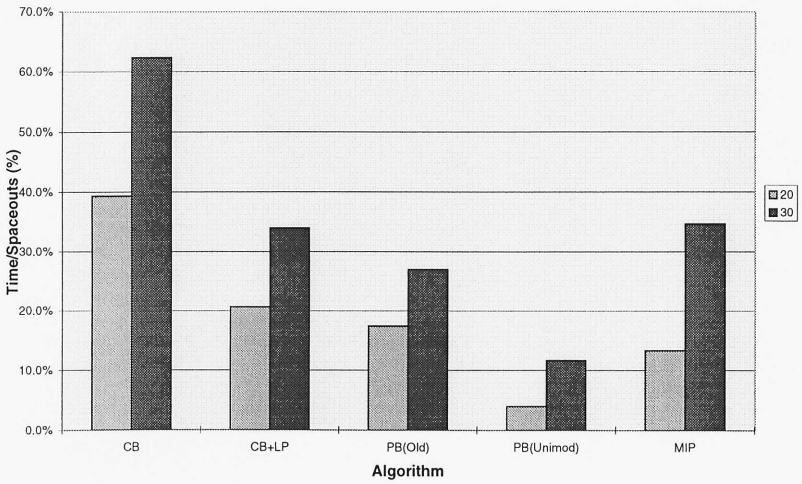


Figure 6.4: Time/spaceout percentages for 20 & 30 task problems (problem set 1)

Since the first four algorithms ran on the ECL^{i}PS^e platform, and shared much of their code, time/spaceouts are a good means of comparison. However, the fifth, MIP, is a commercial product that is implemented differently. To obtain a fairer assessment, the two algorithms MIP and PB(Unimod) were compared on the problems in

set 1 that were solved to optimality by *both*. They were compared by average CPU time, as well as the average number of times they invoked the linear solver (number of LP nodes). Notice the number of LP nodes is identical to number of search nodes visited, since the linear solver is invoked once at each search node. Table 6.2 gives the averages together with 90th percentiles.

Table 6.2: Problem set 1: MIP vs. PB(Unimod), on problems solved to optimality by both

Algorithm	LP Nodes Avg. / 90th percentile	CPU time (s) Avg. / 90th percentile
PB(Unimod)	**583 / 1148**	**50 / 92**
MIP	25636 / 72757	173 / 591

As expected, the specialized commercial package MIP manages many more Simplex calls per second than the PB(Unimod) implementation (an average of 148 calls per second, vs. only 12). Despite this, the results show that PB(Unimod) reduces the number of LP nodes to the extent that it obtains a significant CPU time advantage.

The next problem set, problem set 2, was composed of 1000 problems. The aim was to investigate the performance of the algorithms on a wider range of problems, involving larger resource reductions and temporal shifts. The no. of activities was fixed at 20, and the expected density E_{density} at 0.5. The ranges attempted for time shift and resource reduction were respectively{$\pm$20,$\pm$50,$\pm$100,$\pm$500,$\pm$1000} and {-1,-3,-5,-7}. Note that the largest temporal shift does not constrain the temporal variables at all, allowing activities to be placed anywhere on the scheduling horizon. 50 problem instances were tested at each sample point.

Table 6.3: Problem set 2: Search results

Algorithm	Time/Spaceouts	Proofs(Optimal)	Proofs(Infeasible)
CB+LP	21.5%	22.8%	55.7%
CB+LP+Edge	17.4%	23.0%	59.6%
PB(Unimod)	16.4%	26.4%	57.2%
PB(Unimod)+Edge	13.1%	26.5%	60.4%
PB(Unimod)+Edge+Alt	**12.2%**	**27.6%**	**60.2%**
MIP	27.6%	21.6%	50.8%

The five algorithms tested were CB+LP, CB+LP+Edge, PB(Unimod), PB(Unimod)-+Edge, PB(Unimod)+Edge+Alt, and MIP. The results show that the PB(Unimod) algorithms maintain their advantage for this wider problem set. They show also that both CB+LP and PB(Unimod) are improved by the addition of edge finding propagation because it expedites proofs of infeasibility. This seems to indicate that the addition of advanced propagation methods will yield similar benefits to both types of algorithm. A performance improvement was also noted with the use of the alternative secondary heuristic Alt.

The final and largest series of experiments, problem set 3, tested how the performance advantage of PB(Unimod) varied with the hardness of the problem. The following parameters were applied: no. of activities ($\{20\}$); E_{density} ($\{0.5\}$); maximum time shift ($\{\pm100,\pm200, \ldots ,\pm1000\}$); and number of resource reductions ($\{-1,-2, \ldots ,-10\}$). 100 instances were tested at each point, to make a total of 10,000 problems.

Representatives of the three different algorithm classes, MIP, CB+LP+Edge, PB(Unimod)+Edge+Alt were compared on this much larger problem set. Figures 6.5, 6.6 and 6.7 show their time/spaceouts distributed over the maximum time shift and resource reduction parameters. The graphs show the performance gains of PB(Unimod)+Edge+Alt are sustained as the hardness of the problem increases.

6.9 Conclusion

6.9.1 Discussion

The resource utilization problem shows up very clearly the relative advantages (and disadvantages) of constraint programming (CP) and mathematical programming. CP algorithms are suited to the satisfaction of disjunctive constraints, but local

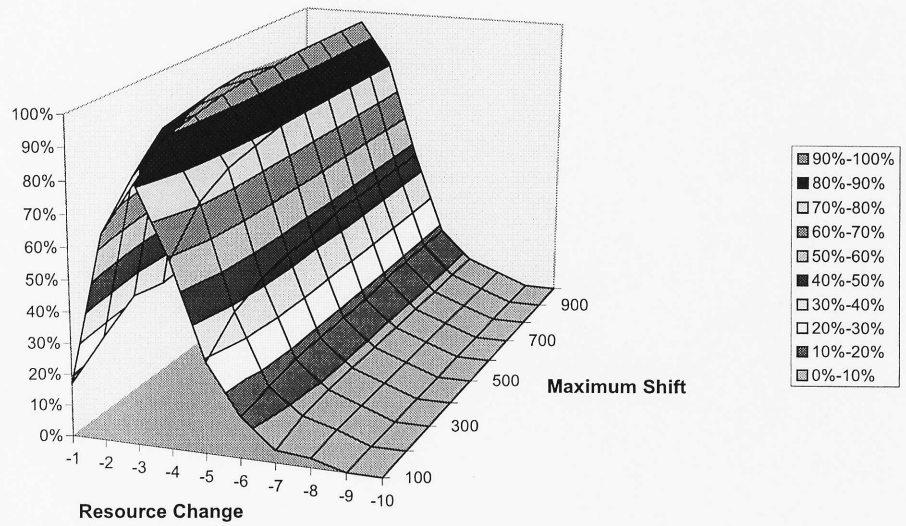


Figure 6.5: MIP time/spaceouts versus resource reduction and max. time shift (problem set 3)

heuristics and consistency methods are generally not effective at global optimization. In mathematical programming, the optimization function is central to the search, but disjunctive constraints are hard to satisfy.

The results of Section 6.8 demonstrate that the unimodular probing strategy is capable of proving optimality where conventional hybridizations of CP backtrack search fail to do so. The dramatic reduction in the number of LP solver calls by unimodular probing as opposed to MIP is also worthy of note.

The reasons for improvement over conventional CP backtrack search algorithms are more or less clear. The probe backtracking algorithm PB focuses the backtrack search on a dynamically changing, high-quality but only partially consistent assignment. Traditionally, local heuristics are constructed by the constraint programmer to reflect the optimization criterion. However, the unimodular probes have greater

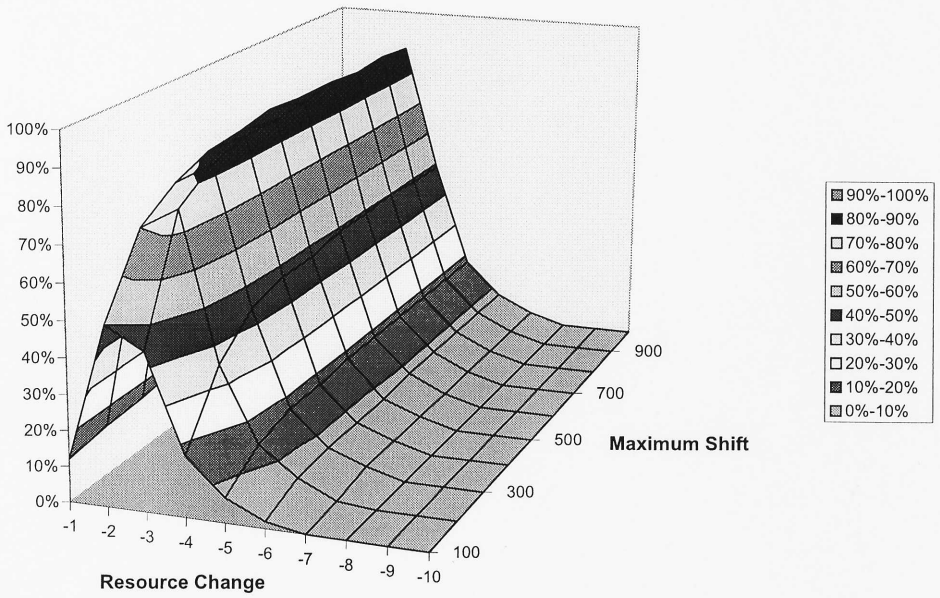


Figure 6.6: CB+LP+Edge time/spaceouts versus resource reduction and max. time shift (problem set 3)

information content, because they are a product of global optimization process. They also already satisfy a significant number of the problem constraints.

When compared to mathematical programming methods in general and MIP search in particular, unimodular probing appears to offer several advantages. First, local consistency methods are applied in unimodular probing, achieving a comparative reduction in the search space. Second, unimodular probing's search ordering heuristics fair better than MIP's. MIP sees all violations in terms of non-integrality in the problem's many integer variables, and thus typically faces a large set of very similar violations during search. Unimodular probing sees instead fewer, more heterogeneous violations of the problem's n-ary constraints. Third, the non-discrete values returned by LP for the relaxed problem are ambiguous in a discrete context, and MIP heuristics based on them may not lead to well informed search choices. In

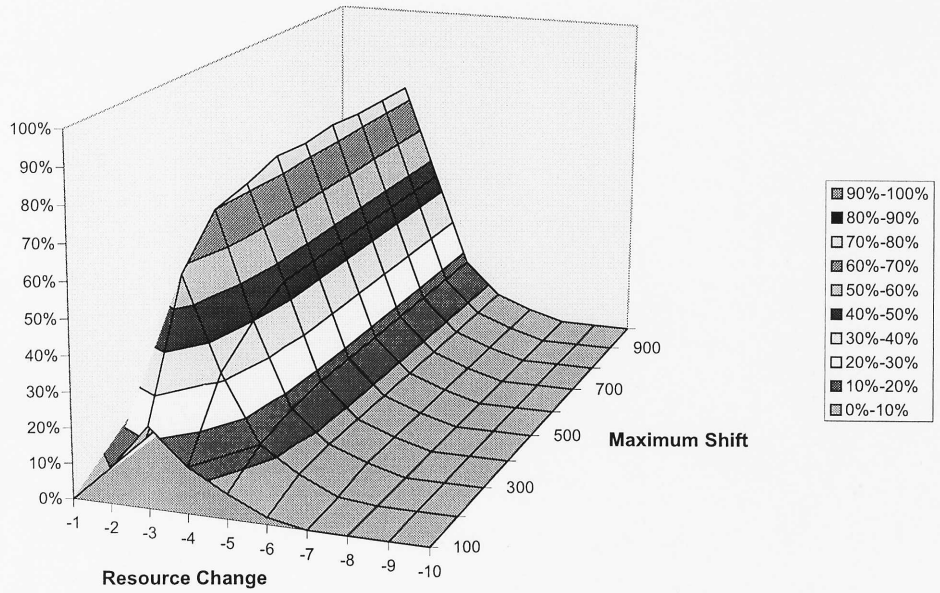


Figure 6.7: PB(Unimod)+Edge+Alt time/spaceouts versus resource reduction and max. time shift (problem set 3)

constrast, the discrete probes of unimodular probing give an accurate view of the sources of constraint violation, enabling the use of semantically clear heuristics for bottleneck prioritization and repair selection (the AI approach).

In terms of its wider applicability, the unimodular probing algorithm can be applied to discrete problems, with a linear optimization function and a TU constraint subset that includes the optimization variables. A newly reported result demonstrates that unimodular probing can yield performance benefits on other problems.[13]

[13]Results of good performance on a chemical processing problem with a just-in-time optimization function have recently been presented [59].

6.9.2 Related Work

I. Probe Backtracking (PB)

Probe backtracking aids backtrack search by utilizing a **forward search** method in addition to the more conventional local consistency checking procedures. This hybridization form is a generalization and a synthesis of several ideas that were implicit in earlier constraint and mathematical programming algorithms:

Static Probe Backtracking *Driving and limiting backtrack search by the conflicts in an initial, partially-consistent assignment.* Early methods used heuristic measures to detect problem bottlenecks and areas of constrainedness (e.g. [79] in scheduling). Minton et al.'s approach used an initial, partially feasible, complete assignment in conjunction with the min-conflicts heuristic [64]. The initial assignment was produced using a polynomial-time procedure that tended to reduce the number of constraint violations. One of the algorithms presented was a backtrack algorithm that repaired regions of constraint violation in the complete assignment. Similar dual assignment methods also employed the notion of partially consistent, tentative assignments to uninstantiated variables [93, 98]. These methods also started from a partially consistent assignment and reduced constraint violations until global feasibility was obtained. Walton used probabilistic analysis to show that a random probe assignment obtained before search improves the average case performance of backtracking (the term *probe* used here is from Walton) [73].

Dynamic Probe Backtracking *Utilizing an independent algorithmic component to update the probe assignment after each search decision.* By this definition, the algorithms referred to above are not dynamic. However, dynamic probe backtracking is to be found implicitly in mathematical programming techniques such as Mixed Integer Programming (MIP) that conduct search by repairing up-to-date linear pro-

gramming probes.

Easy Sub-Problem Probes *Partitioning of the problem requirements into hard and easy sub-problems for handling by separate algorithmic components.* An early constraint programming example is given in [20]. This describes an algorithm that solves an easy sub-problem to elicit information that is used to guide searches of the problem as a whole. The easy sub-problems in this case are backtrack-free portions of the constraint graph. Another example is given by mathematical programming methods that use solutions to the problem's linear relaxation to guide search.

Super-Optimal Probes (optimization only) *Optimizing by repairing probes that are super-optimal with respect to the optimization function.* Again this idea exists in MIP and other methods applying linear programming to relaxed problems.

II. Unimodular Probing (PB(Unimod))

Probe backtracking instantiated to unimodular probing is closely related to MIP. However, unlike MIP, where only unary disjunctive constraints are allowed (the integrality constraints), n-ary disjunctive constraints are represented and monitored for violation.

Several previous approaches have integrated CP methods tightly with LP, creating different styles of hybridization. Beringer and de Backer have simply used constraint propagation & AI learning methods during search to aid linear solvers when they are applied to a disjunct of linear problems [5]. In another approach, LP has been integrated with CP backtrack search and constraint propagation to create an enhanced MIP-style search [77]; however, like MIP, the non-integrality of LP values is used to direct search decisions. Other LP-based frameworks have proposed the expression and repair of n-ary disjunctive constraints, as in unimodular probing

[48]. Yet these frameworks have assumed that the linear solver is applied to the linear relaxation of the problem constraints, and have focused instead on making this relaxation as "tight" as possible. Here also, integrality is the key disjunctive constraint that requires repair during search.

By contrast to all these approaches, unimodular probing dynamically scopes the linear solver to handle sets of constraints having TU. All algorithms that apply LP to the linear relaxation, and seek to associate discrete variables with potentially non-integer values, face the problem of ambiguous interpretation (referred to earlier). In unimodular probing, the source of ambiguity is eliminated, leading to improved algorithm performance.

6.9.3 Summary

This case study described algorithms for minimizing perturbation in dynamically changing schedules. Several algorithms were designed to address the perturbation issue for a generic scheduling problem, the KRFP. One algorithm, unimodular probing, easily outperformed its rivals, as evidenced by a detailed empirical study.

Unimodular probing dynamically scopes linear programming, and incorporates its outputs into backtrack search using probe backtracking, a versatile hybridization strategy. Probe backtracking can be used to integrate other "forward search" methods, and may well have an important role to play in other application domains. This point will be discussed further in the following chapter.

CONCLUSION

Chapter 7

Review of Hybridization

In Chapter 3 we described the conditions that established the hybridization context for the thesis. We then detailed three case studies describing specific algorithms within this context.

We now seek to extract from these hybrid algorithms wider *hybridization forms*: the term "hybridization form" describes a hybridization technique that is general enough to integrate more than just one set of methods. The aim is to support the reader in re-applying these hybridization techniques for other problems and algorithm configurations.

The following sections will detail the hybridization forms of the three case studies in turn, and are all structured as follows:

1. a brief description for the hybridization form in question;
2. a discussion of the hybridization issues that must be addressed;
3. a review of the form's wider potential and limitations, with reference to the relevant case study.

7.1 Form I: Adaptive Propagation

7.1.1 Description

The first case study detailed a hybrid algorithm, anti-functional reduction (AFR), that dynamically hybridized arc consistency and forward checking for binary constraint satisfaction.

More generally, the algorithm demonstrated that it was possible to dynamically scope a higher- and a lower-order consistency method, such that efficiency gains are achieved. Specifically, the scope of the higher-order consistency method is dynamically reduced to exclude parts of the problem where its computational cost/benefit characteristics seem unfavourable. Thus:

Hybridization Form I *is the dynamic scoping of a higher- and a lower-order consistency method, such that: the scope of the higher-order consistency method is initially the whole problem; this scope is reduced dynamically during search to exclude parts of the problem where the lower-order consistency method achieves similar or identical levels of consistency; and, finally, the scope of the lower-order consistency method is expanded in its place.*

7.1.2 Hybridization Issues

Selection & Scoping

This hybridization form is not possible without the identification or development of the following problem- and algorithm-specific components:

- Two consistency methods achieving different orders of consistency that are

appropriate to the problem undergoing solution.

- A dynamic scoping procedure for the higher-order consistency method that reduces its scope at run-time, allowing the lower-order consistency method to take its place.

To construct an effective hybrid algorithm using this form, the two consistency methods should be specially selected to *maximize the opportunities for fruitful dynamic scoping.* Specifically, the higher-order consistency method should:

1. be significantly more (time or space) expensive but no more effective than the lower-order consistency method on parts of the problems addressed;
2. and maintain data structures that allow the identification of these parts.

Moreover, the average-case time complexity of the dynamic scoping procedure should be no greater than the average-case reduction in the time spent checking consistency.

Synchronization & Integration

There is an integration issue concerning the dynamic scoping of consistency methods. That is to say that the dynamic scoping procedure must ensure that every constraint is checked for violation before a solution is output. Failure to do so will lead to loss of soundness. By guaranteeing that every constraint is covered by (at least) one of the two consistency methods, soundness is preserved.

7.1.3 Potential and Limitations

Potentially, this form could be applied to combine other consistency methods.

However, while our experience on AFR highlighted the benefits of the hybridization form, it also showed its limitations. In Section 3.3.1, we stated that dynamic scoping strategies could lead to tighter, and thus more efficient hybrid algorithms, but would require more complicated hybridization processes. AFR clearly illustrates this point. Firstly, dynamically changing scopes justified the associated overhead in AFR, and were the source of its efficiency. Secondly, the dynamic scoping procedures were highly specific to the arc consistency method (AC-4): in fact, the practical efficiency of the AFR algorithm crucially relied on re-using AC-4's dynamic data structures for scoping.

Thus AFR exploited the simplicity and locality of the two consistency methods in question, and the run-time data maintained by AC-4. Similar opportunities must be found if consistency methods maintaining higher-orders of consistency (e.g. path consistency) are to be hybridized in the same way. Since higher-order consistency methods are more complex, it is anticipated that their scoping strategies will also be. However, if effective scoping procedures are found, the potential gain from hybridization is greater, since higher-order consistency methods are computationally more expensive.

An analogous argument applies when the complexity of the addressed problem is increased, for instance, by moving from binary to n-ary CSPs.

7.2 Form II: Sub-Problem Delegation

7.2.1 Description

Case Study II investigated fleet assignment. The hybrid algorithms modified the behaviour of the underlying backtrack algorithm more dramatically than in the first hybridization form. The hybridization technique used a forward search method to

label a subset of the variables (the set ON of overnight variables) in a single "search step", rather than one at a time as in the standard backtrack search procedure.

Naturally the number of alternative labellings at this search step is potentially very large.[1] The role of the forward search method is to generate as few alternatives as possible, while preserving the completeness of the algorithm as a whole. Additionally, the alternatives should be generated in an order that ensures good solutions are found early on in the search.

In the fleet assignment application, the forward search method labels its variables before backtrack search labels the remaining variables. In general, though, the forward search method could be applied at any point in the labelling routine.

Let us extend this simple technique to a general hybridization form. Consider a backtrack search node and the set of unassigned variables at that search node. Let an *exhaustive* forward search method be one that can label a subset of the unassigned variables, in all the different ways that could be extended to a full solution.

Hybridization Form II *is the transfer, at some backtrack search node, of a subset of the unassigned variables for assignment by an exhaustive forward search method, such that algorithm completeness is preserved.*

7.2.2 Hybridization Issues

Selection & Scoping

This hybridization form depends entirely on the appropriate selection and scoping of an exhaustive forward search method. If the hybridization form is to be effective,

[1] In the worst case, d^n alternatives for labelling n variables, where d is the variables' domain size.

this forward search method must improve backtrack search in one or more of the following possible ways:

1. by generating, at earlier stages in the search, labellings that are likely to be extended to solutions;[2]
2. by generating earlier labellings that are better with respect to the optimization criteria;
3. or by labelling the set of variables more efficiently than a corresponding backtrack search procedure.

Synchronization & Integration

The issue of how to integrate the forward search method, such that soundness and completeness are preserved, is dealt with straightforwardly. The hybrid algorithm does not threaten soundness if the labels generated by the forward search method are checked for consistency. Completeness is also retained, if *all* the feasible alternative labellings are generated in turn by the forward search method, in a process that is analogous to the assignment of a single variable in ordinary backtrack search.

7.2.3 Potential and Limitations

The hybridization form is quite straightforward to implement and has broad potential applicability. Any backtrack search algorithm could be modified to delegate (at some point during search) the responsibility for labelling a subset of the variables

[2]Some assignment orderings cannot be generated by ordinary backtrack search without modifications to the algorithm or the problem. For example, suppose we are labelling two variables X and Y with domains $\{1, 2\}$. An ordinary backtrack search algorithm is not capable of producing the following assignment ordering: (1) $X = 1, Y = 2$; (2) $X = 2, Y = 1$; (3) $X = 1, Y = 1$; (4) $X = 2, Y = 2$.

to an exhaustive forward search method. These variables may not be well suited to labelling by ordinary backtrack search methods, because of the nature of the constraints they are subject to, or because of their role in the optimization function.

However, the creation of efficient and yet complete hybrids using this hybridization form depends on the success of two steps. Step 1 is the identification of those sets of variables suitable for delegation. Step 2 is the construction of appropriate, exhaustive, forward search methods for labelling them. The use of exhaustive methods is important, not only to create complete hybrids, but also to avoid the pitfalls normally associated with incomplete algorithms (viz. the tendency to re-visit previously explored search nodes, and to get trapped in local minima).

In Case Study II, these two steps were successfully addressed.[3] It was clear from the structure of the problem that the "overnight" variables would be labelled better by a *repair-based* method that (a) chose values that focused search on regions close to the airline's current assignment, and (b) broke up the problem into relatively independent sub-problems that could be solved more easily by backtrack search. Moreover, simple forward search methods based on LDS were developed to conduct repair-based (and local) searches that were exhaustive and yet space efficient.

To conclude, our experience on fleet assignment hints that, at least for structured LSCOs, the identification of suitable subsets of variables may not be problematic (Step 1). The construction of exhaustive forward search methods is also straightforward (Step 2). The LDS and HLDS procedures presented are very general, and show, for small-sized sub-problems, that it is realistic to create exhaustive repair-based or local search methods. In principle, similar procedures could be utilized in other problems.[4]

---

[3]The deployment of "exhaustive" LDS approaches in Case Study I meant that the basic hybrid algorithms presented were complete (before time-outs lost this property).

[4]As we suggested in Section 5.6.3, yet more sophisticated methods could be created by replacing the hill climbing neighbourhood operator used in HLDS: for example, by the temperature-controlled operators of *simulated annealing* [54].

7.3 Form III: Probe Backtracking

7.3.1 Description

Case Study III explored algorithms for minimal perturbation in dynamic scheduling. In the course of the case study, the third hybridization form — designated probe backtracking — was introduced.

In this form, backtrack search is supported by a *probe generator*. This is a forward search procedure that generates good quality, but only partially consistent assignments (*probes*). Each unassigned variable is associated not only with a domain of values, but also with its tentative value according to the probe. The tentative value is not necessarily adhered to and may be changed by the probe generator as the backtrack search progresses. The purpose of the probe generator is to direct the backtrack search component and *limit the size of its search space*: the backtrack search is focused only on regions where the probe violates constraints, and search terminates as soon as the constraint violations are eliminated from the probe.

According to this hybridization form the problem constraints are divided into two sets: the set of constraints that are always satisfied by the probe (*probe constraints*), and the set of constraints that may be violated by it (*monitored constraints*). Monitored constraints are so called because they are monitored during search for violation. At each search node, a violated constraint is selected, and a backtrack-able decision is then made by the backtrack search, to eliminate the possibility of the same violation recurring in subsequent probes. Hence this decision must be representable as a probe constraint.[5] On backtracking, an alternative probe constraint is posted, such that the search space is fully partitioned and exhaustively explored.

[5]Recall that in backtrack search, decisions may be made via the posting of arbitrary constraints, and not necessarily via variable assignments (as indicated in Section 2.2, and exemplified by Case Study III).

We may summarize this hybridization form as follows:

Hybridization Form III *is: the application, at every backtrack search node, of the forward search method to generate a* ***probe****; the posting of a backtrack-able* ***probe constraint*** *that eliminates one of the sources of constraint violation in the* ***probe****; and the posting of other* ***probe constraints*** *on backtracking, such that a complete search results.*

7.3.2 Hybridization Issues

Selection & Scoping

The first step in creating hybrids using this form, is to select and appropriately scope a probe generator. For efficient constraint *satisfaction* search, the probe generator should ideally:

1. be time and space efficient;
2. and satisfy as many of the constraints as possible (maximize the set of probe constraints, thus minimizing the monitored constraints that must be repaired through backtracking).

Thus the hybridization form can support constraint satisfaction by focusing search on repairing partially consistent assignments with a high degree of consistency.

However, *for optimization, the probe generator must additionally be* ***super-optimal.***[6] This ensures the optimization process is complete, as explained in the next section.

[6] A super-optimal assignment is a partially-consistent assignment that is at least as good as the feasible optimal solution with respect to the optimization criteria (Section 6.6.2).

With respect to scoping, in some cases a dynamic scoping strategy may be used to improve the efficiency of the probe generator — this was the case in the unimodular probing algorithm of Case Study III.

Synchronization & Integration

In this hybridization form, it is necessary to monitor for violation all constraints that are not guaranteed to be satisfied by the probe. Thus the problem constraints must be partitioned fully into the set of probe constraints and the set of monitored constraints. Otherwise, algorithm soundness would be jeopardized by the fact that it terminates as soon as the monitored constraints are satisfied by the latest probe.[7]

Ensuring algorithm completeness is more complex. The minimal requirements, for probe backtracking algorithms applied to *finite* constraint satisfaction (and optimization) problems, have been listed below without proof.

1. The probe generator must be sound and complete in the satisfaction sense, and trigger backtracking when infeasibility is detected in the posted probe constraints.

2. In the case of optimization, the probe generator must always generate super-optimal probes. Otherwise, search may terminate when feasible sub-optimal solutions are found.

3. On selecting a violated (monitored) constraint for repair, the backtrack algorithm must post a new *repair* probe constraint, that ensures subsequent probes in this search branch do not return the same values for the violated

[7]N.B. Case Study III described how one of the probe backtracking algorithms (PB(Old)) failed to satisfy this requirement, which is why an extension to the algorithm was necessary for soundness (Section 6.7.2).

constraint's variables.[8]

4. On backtracking, the backtrack search must post probe constraints that preserve completeness (possibly after alternative repair probe constraints have been attempted). Let k be the total number of alternative probe constraints posted at a given search node. These must be such that the k sub-problems, corresponding to the k different search branches, contain all the input problem's solutions. In other words, the union of the solution sets of the k sub-problems is equal to the solution set of the search node's initial problem.[9]

7.3.3 Potential and Limitations

Case Study III described how this hybridization form was used to improve efficiency. The case study detailed a progression of hybrid algorithms from traditional "loose" hybridizations to the very tight hybridizations achieved by this form. The empirical study showed that the quality of performance was proportional to the tightness of hybridization. Moreover, to our knowledge, unimodular probing — one of the algorithms applying the hybridization form — is currently the most efficient algorithm in existence for the class of problems addressed.[10]

From a wider perspective, this very general hybridization form can yield efficient search algorithms for many problems unconnected to dynamic scheduling or minimal perturbation.[11] In our view, the full potential of the hybridization form has yet to be discovered.

---

[8] For instance, if the probe generator satisfies only domain bounds, search can progress only by narrowing these bounds, i.e. by variable assignments or domain splitting.

[9] In the case study $k = 2$, since temporal constraints and their negations were posted.

[10] It has also successfully passed commercial performance testing at British Airways, where it is currently in service.

[11] Unimodular probing, an instance of this form, has already been applied to several process design benchmarks where it outperformed rival constraint and mathematical programming algorithms [59].

However, in the application of this form, the constraint programmer is faced with the task of constructing a fast and effective probe generation procedure, satisfying as many constraints as possible (and producing super-optimal assignments, if a sound and complete optimization algorithm is required).[12] Yet one should take heart from the result by Walton [73], which shows that (even) a randomly-generated probe that does not change throughout the search, improves the average-case performance of the backtrack search algorithm.

Moreover, the fact that the probe generator needs neither to be exhaustive (in contrast to Hybridization Form II), nor to satisfy all the problem constraints, means that a wealth of existing algorithms are candidate generators. Section 6.9.2 remarked that existing mathematical programming methods implicitly integrated linear solvers with backtrack search in this way. The form could also be used to hybridize local search algorithms such as hill climbing, simulated annealing, and genetic programming methods. Conventionally viewed as substitutes for backtrack search, such algorithms can now act in support of it.

7.4 Future Research

In constraint programming circles the drawbacks of backtrack search have been exhaustively explored and commented upon. Yet ordinary backtrack search resiliently remains the workhorse of the constraint programming paradigm, due to its simplicity and versatility.

Much work has suggested enhancements of the basic backtrack search algorithm. These have chiefly sought to avoid the execution of large backtrack searches by developing more sophisticated search heuristics, consistency methods or learning &

[12] For hard problems, super-optimality could be abandoned in favour of good quality probes, sacrificing theoretical soundness and completeness in exchange for improved average-case performance.

intelligent backtracking techniques.

Such enhancements go hand in hand with the goal pursued here, namely that of dynamic and localized hybridization. Every backtrack search node is a potential point at which various "support methods" may be selected, scoped, and applied, before their results are incorporated into the backtrack search procedure. Naturally, shared and dynamically changing responsibilities among the various algorithm components complicate hybridization: scoping rules may need to be automated, and results integrated at run-time. Yet, as the case studies have shown, in many cases the tightly hybridized algorithms that emerge justify the effort invested.

Looking to the future, our proposal that *forward search* be considered as an alternative enhancement category for backtrack search opens new avenues for investigation. For example, in addition to the creation of specialized consistency methods for global constraints (such as the cumulative constraint of CHIP [2]), one might now also envisage the development of forward search methods specialized to solve them. The aim would be to integrate them by Hybridization Forms II or III.

Certainly, the proliferation of forward search methods would burden the backtrack search algorithm with entirely new tasks. Under the proposal, the role of backtrack search is extended beyond the simple execution of search, to include also the delegation of scoped sub-problems to forward search methods, and the reconciliation of their results when they are contradictory or inconsistent. The investigation of such search "management" activities is a rich domain of research.

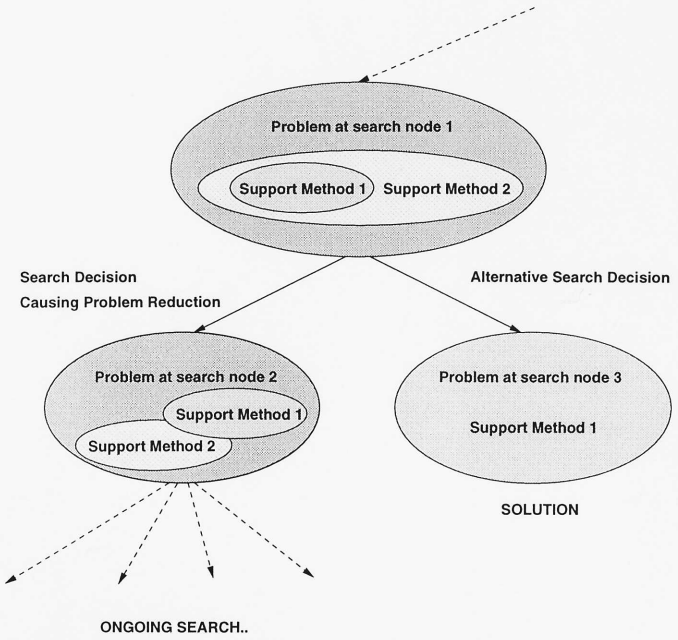


Figure 7.1: Search management: the future for backtrack search?

Chapter 8

Summary of Contributions

The thesis presents three case studies exploring different problems. Each case study describes an original hybrid algorithm within our chosen hybridization context to enhance the efficiency of backtrack search. A conclusion reviews the forms of hybridization the algorithms exemplify. The specific contributions are listed below.

Case Study I: Adaptive propagation

1. The concept of adaptive propagation, whereby automated scoping rules hybridize a higher-order consistency method with a lower-order one such that efficiency improvements are obtained.

2. The anti-functional reduction (AFR) algorithm, which enhances the arc consistency algorithm AC-4, by hybridizing it with forward checking (FC).

3. An empirical comparison of AFR, AC-4 and FC on randomly-generated, binary constraint satisfaction problems. The results confirm that AFR achieves performance improvements over its component algorithms.

Case Study II: Sub-Problem Delegation

1. The development of a backtrack search algorithm for fleet assignment, enhanced by specially selected search ordering heuristics, and the application of redundant constraints.

2. A hybridization technique that integrates repair-based algorithms into the backtrack search algorithm, to increase its optimization efficiency while preserving completeness.

3. The development of HLDS, an algorithm that combines hill climbing with limited discrepancy search (LDS). The demonstration, by HLDS, of how LDS procedures can be used as neighbourhood operators within local search, to create complete search algorithms with the space-complexity of backtrack search.

4. An experimental study of two hybrids utilizing the hybridization technique, the first integrating LDS, and the second HLDS, into backtrack search. The hybrids are compared with two conventional backtrack search algorithms on four commercial fleet assignment problems. The study shows that, for small problems with few fleets and stations, little or no improvements are obtained by hybridization. However, with increases in the number of fleets and stations, the advantage of the hybridization technique becomes increasingly apparent.

Case Study III: Probe Backtracking

1. The formulation of the minimal perturbation problem, the kernel resource feasibility problem (KRFP),[1] and the resource utilization problem (RUP).

2. A proof that, for the perturbation function considered, Simplex-based methods optimize the temporal sub-problem of the KRFP while ensuring that the temporal variables are assigned only discrete (integer) values.

[1] Based on the resource feasibility problem of planning [24].

3. The hybridization of an existing resource feasibility procedure, with a linear programming solver applied to the temporal sub-problem, to create the first hybrid algorithm for minimal perturbation scheduling (CB). The modification of this algorithm to obtain improved cost bounds via the application of linear programming during the resource feasibility stage. The application of edge-finding for improved consistency checking.

4. The definition of probe backtracking, a generic form for hybridization, and its instantiation for solving the KRFP.

5. The naive probe backtracking algorithm PB(Old) and its improved cost bound and edge-finding variants.

6. The improved probe backtracking algorithm, unimodular probing PB(Unimod). Its heuristic and edge-finding variants. To our knowledge, unimodular probing is the best-performing algorithm in existence today for the class of problems investigated. It is currently in commercial use.

7. The empirical study of the above algorithms and mixed integer programming MIP on 12,800 randomly-generated resource utilization problems, divided into three different problem sets. The study establishes that the unimodular probing algorithm outperforms all other tested algorithms across the three problem sets, maintaining its advantage for hard problems.

Review of Hybridization

A review of the three hybridization forms utilized in the case studies, with focus on associated hybridization issues, and the potential for re-application.

References

[1] Abara, J. Applying integer linear programming to the fleet assignment problem. *Interfaces 19*, 4 (1989), 20–28.

[2] Aggoun, A., and Beldiceanu, N. Extending CHIP in order to solve complex scheduling and placement problems. *Premières Journées Francophones sur la Programmation en Logique* (1992).

[3] Barnhart, C. Flight string models for aircraft fleeting and routing. *Transportation Science 32*, 3 (1998), 208–220.

[4] Bartusch, M., Möhring, R. H., and Radermacher, F. J. Scheduling project networks with resource constraints and time windows. *Annals of Operations Research 16* (1988), 201–240.

[5] Beringer, H., and de Backer, B. Satisfiability of boolean formulas over linear constraints. In *Proc. of International Joint Conference on AI (IJCAI)* (Chambéry, France, August 1993), pp. 296–301.

[6] Bessière, C. Arc-consistency and arc-consistency again. *Artificial Intelligence 65* (1994), 179–190.

[7] Bessière, C., Freuder, E. C., and Régin, J. Using inference to reduce arc consistency computation. In *Proc. of International Joint Conference on AI (IJCAI)* (Montréal, August 1995), pp. 592–598.

[8] Bessière, C., and Régin, J. Using bidirectionality to speed up arc-consistency processing. In *Constraint Processing* (1995), pp. 159–169.

[9] Bland, R., Goldfarb, D., and Todd, M. The ellipsoid method: A survey. *Operations Research 29* (1981), 1039–91.

[10] Borrett, J. E., Tsang, E., and Walsh, N. Adaptive constraint satisfaction: The quickest first principle. In *Proc. of European Conference on AI (ECAI)* (Budapest, Hungary, 1996).

[11] Clarke, L., Hane, C., Johnson, E., and Nemhauser, G. Maintenance and crew considerations in fleet assignment. *Transporation Science 30*, 3 (1996), 249–260.

[12] Cooper, M. C. An optimal k-consistency algorithm. *Artificial Intelligence 41* (1989), 89–95.

[13] Dantzig, G. B. *Linear Programming and Extensions.* Princeton university Press, 1963.

[14] Davenport, A. Managing uncertainty in scheduling: a survey. Working Draft, 1998.

[15] Davenport, A., Tsang, E., Wang, C., and Zhu, K. Genet: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. *Proc. of National Conference on AI (AAAI)* (1994), 325–330.

[16] Dechter, R. Enhancement schemes for constraint processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence 41* (1990).

[17] Dechter, R., and Dechter, A. Belief maintenance in dynamic constraint networks. In *Proc. of National Conference on AI (AAAI)* (1988), pp. 37–42.

[18] Dechter, R., and Meiri, I. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence 68* (1994).

[19] Dechter, R., Meiri, I., and Pearl, J. Temporal constraint networks. *Artificial Intelligence 49* (1991), 61–95.

[20] Dechter, R., and Pearl, J. Network-based heuristics for Constraint Satisfaction Problems. *Artificial Intelligence 34* (1988).

[21] Deville, Y., and Van Hentenryck, P. An efficient arc consistency algorithm for a class of CSP problem. In *Proc. of International Joint Conference on AI (IJCAI)* (Sydney, August 1991), pp. 325–330.

[22] Dincbas, M., Simonis, H., and Van Hentenryck, P. The constraint logic programming language CHIP. In *FGCS* (Japan, 1988).

[23] El-Kholy, A. *Resource Feasibility in Planning.* PhD thesis, Imperial College, University of London, 1996.

[24] El-Kholy, A., and Richards, B. Temporal and resource reasoning in planning: The *parc*PLAN approach. In *Proc. of European Conference on AI (ECAI)* (Budapest, Hungary, 1996), W. Wahlster, Ed., pp. 614–618.

[25] El Sakkout, H. Modelling fleet assignment in a flexible environment. In *Proc. of Int. Conf. on Practical Applications of Constraint Technology (PACT)* (London, UK, 1996), The Practical Applications Co. Ltd.

[26] El Sakkout, H., and Novello, S. The Repair library. In *the ECL^{i}PS^e Libraries Manual.* 1997.

[27] El Sakkout, H., Richards, T., and Wallace, M. Unimodular probing for minimal perturbance in dynamic resource feasibility problems. In *Proc. of the CP'97 workshop on Dynamic Constraint Satisfaction* (Linz, Austria, 1997).

[28] El Sakkout, H., Richards, T., and Wallace, M. Minimal perturbation in dynamic scheduling. In *Proc. of European Conference on AI (ECAI)* (Brighton, UK, 1998).

[29] El Sakkout, H., and Wallace, M. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints. Special Issue on Constraint-Directed Reasoning Issues Arising From Industrial Scheduling Applications* (2000). To appear.

[30] El Sakkout, H., Wallace, M., and Richards, B. An instance of adaptive constraint propagation. In *Proc. of Int. Conf. on Principles & Practice of Constraint Programming (CP)* (Boston, 1996).

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

[32] French, S. *Sequencing and Scheduling.* Ellis Horwood Limited, 1990, ch. 1-2.

[33] Freuder, E. C. Synthesizing constraint expressions. *Communications of the ACM 21*, 11 (November 1978), 958–966.

[34] Freuder, E. C. Using metalevel knowledge to reduce constraint checking. In *Constraint Processing: Selected Papers from the ECAI'94 Workshop.* Springer, 1995.

[35] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Bell Telephone Laboratories, Inc., 1979.

[36] Garfinkel, R. S., and Nemhauser, G. L. *Integer Programming.* John Wiley & Sons, 1972.

[37] Gaschnig, J. Performance measurement and analysis of certain search algorithms. Tech. rep., Carnegie-Mellon University, 1979.

[38] Ginsberg, M. L. Dynamic backtracking. *Journal of Artificial Intelligence Research* (1993), 25–46.

[39] Glover, F. Tabu search Part I. *Operations Research Society of America (ORSA) Journal of Computing 1* (1989).

[40] Glover, F. Tabu search Part II. *Operations Research Society of America (ORSA) Journal of Computing 2* (1990).

[41] Golomb, S. W., and Baumert, L. D. Backtrack programming. *Journal of the ACM 12* (1965), 516–524.

[42] Hane, C., Barnhart, C., Johnson, E., Marsten, R., Nemhauser, G., and Sigismondi, G. The fleet assignment problem: Solving a large-scale integer program. *Mathematical Programming 70* (1995), 211–232.

[43] Haralick, R. M., and Elliot, G. L. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14* (October 1980), 263–314.

[44] Harvey, W., and Ginsberg, M. Limited discrepancy search. In *Proc. of International Joint Conference on AI (IJCAI)* (Montréal, August 1995).

[45] Heller, I., and B.Tompkins, C. An extension of a theorem of Dantzig's. In *Linear Inequalities and Related Systems*, Kuhn and Tucker, Eds. Princeton University Press, 1956, pp. 247–254.

[46] Hildum, D. W. *Flexibility in a Knowledge-Based System for Solving Dynamic Resource-Constrained Scheduling Problems.* PhD thesis, Dept. of Computer Science, University of Massachusetts, Amherst, 1994.

[47] Holland, J. H. *Adaption in natural and artificial systems.* The University of Michigan Press, Ann Arbor, 1975.

[48] Hooker, J. N., and Osorio, M. A. Mixed logical/linear programming. *Discrete Applied Mathematics (to appear)* (1998).

[49] IC-PARC. *ECL^{i}PS^e User Manual*, 1997.

[50] ILOG Inc. *ILOG Solver User Manual*, 1997.

[51] Kanal, L., and Kumar, V. *Search in Artificial Intelligence.* Springer-Verlag, 1988.

[52] Karmarkar, N. A new polynomial-time algrithm for linear programming. *Combinatorica 4*, 4 (1984), 373–395.

[53] Khachian, L. G. A polynomial algorithm in linear programming. *Soviet Math. Dokl. 20*, 1 (1979), 191–194.

[54] Kirckpatrick, S., Gelatt Jr., C. D., and Vecchi, M. P. Optimisation by simulated annealing. *Science 220:4598* (1983), 671–680.

[55] Lawler, E. W., and Wood, D. E. Branch-and-bound methods: a survey. *Operations Research 14* (1966), 699–719.

[56] Le Pape, C., Couronné, P., Vergamini, D., and Gosselin, V. Time-versus-capacity compromises in project scheduling. In *Proc. of the 13th Workshop of the UK Planning and Scheduling SIG* (Glasgow, Scotland, 1994).

[57] Leon, V. J., Wu, S. D., and Storer, R. H. Robustness measures and robust scheduling for job shop. *IIE Transactions 26*, 5 (1994), 32–43.

[58] Lhomme, O. Consistency techniques for numeric CSPs. In *Proc. of International Joint Conference on AI (IJCAI)* (1993), pp. 232–238.

[59] Liatsos, V. Short term scheduling. *Presentation at the DIMACS Workshop on Constraint Programming and Large-Scale Discrete Optimization*, Rutgers University, NJ, 1998.

[60] Mackworth, A. K. Consistency in networks of relations. *Artificial Intelligence 8*, 1 (1977), 99–118.

[61] Mackworth, A. K. The logic of constraint satisfaction. *Artificial Intelligence 58* (1992), 3–20.

[62] Marriott, K., and Stuckey, P. J. *Programming with Constraints: An introduction.* MIT Press, 1998.

[63] Maruyama, F., Minoda, Y., Sawada, S., and Takizawa, Y. Constraint satisfaction and optimization using nogood justification. In *Proc. of the 2nd Pacific rim international conference on Artificial Intelligence 1992* (1992).

[64] Minton, S., Johnston, M. D., Philips, A. B., and Laird, P. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence 58* (1992), 161–205.

[65] Mohr, R., and Henderson, T. C. Arc and path consistency revisited. *Artificial Intelligence 28* (1986), 225–233.

[66] Montanari, U. Networks of constraints : Fundamental properties and applications to picture processing. *Information Sciences 7*, 2 (1974), 95–132.

[67] Nemhauser, G., and Wolsey, L. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.

[68] Nuijten, W. *Time and Resource Constrained Scheduling: A constraint satisfaction approach.* PhD thesis, Eindhoven University of Technology, 1994.

[69] Nuijten, W., and Aarts, E. Constraint satisfaction for multiple capacitated job shop scheduling. In *Proc. of European Conference on AI (ECAI)* (1994), John Wiley & Sons, Ltd.

[70] Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

[71] Pothos, D. A constraint-based approach to the British Airways schedule re-timing problem. Tech. Rep. 97/04-01, IC-Parc, 1997.

[72] Prosser, P. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence 9* (1993).

[73] Purdom Jr., P. W., and Haven, G. N. Probe order backtracking. *Siam Journal of Computing 26* (1997), 456–483.

[74] Queyranne, M., and Wang, Y. Single-machine scheduling polyhedra with precedence constraints. *Mathematics of Operations Research* (1991), 1–20.

[75] Richards, T. *Non-systematic search and no-good learning.* PhD thesis, IC-Parc, 1998.

[76] Richards, T., Jiang, Y., and Richards, B. NG-backmarking - an algorithm for constraint satisfaction. *BT Technol J 13* (1995).

[77] Rodošek, R., and Wallace, M. A generic model and hybrid algorithm for hoist scheduling problems. In *Proc. of Int. Conf. on Principles & Practice of Constraint Programming (CP)* (Pisa, 1998).

[78] Sabin, D., and Freuder, E. C. Contradicting conventional wisdom in constraint satisfaction. In *Proc. of European Conference on AI (ECAI)* (Amsterdam, August 1994), pp. 125–129.

[79] Sadeh, N. Micro-opportunistic scheduling: The micro-boss factory scheduler. In *Intelligent Scheduling*, M. Zweben and M. Fox, Eds. Morgan Kaufman, 1994, ch. 4, pp. 99–136.

[80] Schiex, T., Régin, J., Gaspin, C., and Verfaillie, G. Lazy arc consistency. In *Proc. of National Conference on AI (AAAI)* (1996).

[81] Schiex, T., and Verfaillie, G. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools 3(2)* (1994).

[82] Schrijver, A. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[83] Smith, B. Succed-first or fail first: A case study in variable and value ordering. In *Proc. of Int. Conf. on Practical Applications of Constraint Technology (PACT)* (1997), The Practical Applications Co. Ltd., pp. 321–330.

[84] Smith, B. M., and Grant, S. A. Sparse constraint graphs and exceptionally hard problems. In *Proc. of International Joint Conference on AI (IJCAI)* (Montréal, August 1995), pp. 646–651.

[85] Subramanian, R. Coldstart: Fleet assignment at Delta Air Lines. *Interfaces 24*, 1 (1994), 104–120.

[86] Tong, C. *A Divide-and-Conquer Approach to Knowledge Compilation.* AAAI Press, 1991, pp. 261–287.

[87] Tong, C., Braudaway, W., Mohan, S., and Voigt, K. Reformulating constraints for compilability and efficiency. *Proc. of National Conference on AI (AAAI) Workshop on Problem Reformulation and Change of Presentation* (1992).

[88] Tsang, E. *Foundations of Constraint Satisfaction.* Computation in Cognitive Science. Academic Press, 1993.

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

[90] Van Hentenryck, P. A logic language for combinatorial optimization. *Annals of Operations Research 21* (1989).

[91] Van Hentenryck, P., and Carillon, J. Generality versus specificity: an experience with AI and OR techniques. In *Proc. of National Conference on AI (AAAI)* (Minnesota, August 1988).

[92] Van Hentenryck, P., Deville, Y., and Teng, C. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence 57* (1992), 291–321.

[93] Verfaillie, G., and Schiex, T. Solution reuse in dynamic constraint satisfaction problems. In *Proc. of National Conference on AI (AAAI)* (Seattle, WA, August 1994), pp. 307–312.

[94] Voudouris, C., and Tsang, E. Guided local search. *European Journal of Operational Research 1 13, issue 2* (1999), 469–499.

[95] Wallace, R. J. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proc. of International Joint Conference on AI (IJCAI)* (Chambéry, France, August 1993), pp. 239–245.

[96] Waltz, D. L. Generating semantic descriptions from drawings of scenes with shadows. Tech. Rep. MAC-AI-TR-271, MIT, 1972.

[97] Waltz, D. L. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, P. H. Winston, Ed. McGraw-Hill, 1975, pp. 19–91.

[98] Yokoo, M. Weak-commitment search for solving constraint satisfaction problems. In *Proc. of National Conference on AI (AAAI)* (Seattle, WA, August 1994), pp. 313–318.