Constraint Weighting Local Search for Constraint Satisfaction

by

John Thornton

B.Bus, Griffith University, Australia (1993)

B.Sc (Hons), Griffith University, Australia (1995)

A thesis submitted in fulfillment
of the requirements of the degree of
Doctor of Philosophy

School of Computing and Information Technology
Faculty of Engineering and Information Technology
Griffith University, Australia

January 2000

Abstract

One of the challenges for the constraint satisfaction community has been to develop an automated approach to solving Constraint Satisfaction Problems (CSPs) rather than creating specific algorithms for specific problems. Much of this work has concentrated on the development and improvement of general purpose backtracking techniques. However, the success of relatively simple local search techniques on larger satisfiability problems [Selman et al. 1992] and CSPs such as the *n*-queens [Minton et al. 1992] has caused interest in applying local search to constraint satisfaction. In this thesis we look at the usefulness of constraint weighting as a local search technique for constraint satisfaction. The work is based on the clause weighting ideas of Selman and Kautz [1993] and Morris [1993] and applies, evaluates and extends these ideas from the satisfiability domain to the more general domain of CSPs. Specifically, the contributions of the thesis are:

- **The introduction of a local search taxonomy.** We examine the various better known local search techniques and recognise four basic strategies: restart, randomness, memory and weighting.

- **The extension of the CSP modelling framework.** In order to represent and efficiently solve more realistic problems we extend the CSP modelling framework to include array-based domains and array-based domain use constraints.

- **The empirical evaluation of constraint weighting.** We compare the performance of three constraint weighting strategies on a range of CSP and satisfiability problems and with several other local search techniques. We find that no one technique dominates in all problem domains.

- **The characterisation of constraint weighting performance.** Based on our empirical study we identify the weighting behaviours and problem features that favour constraint weighting. We conclude weighting does better on structured problems where the algorithm can recognise a harder sub-group of constraints.

- **The extension of constraint weighting.** We introduce an efficient arc weighting algorithm that additionally weights connections between constraints that are simultaneously violated at a local minimum. This algorithm is empirically shown to outperform standard constraint weighting on a range of CSPs and within a general constraint solving system. Also we look at combining constraint weighting with other local search heuristics and find that these hybrid techniques can do well on problems where the parent algorithms are evenly matched.

- **The application of constraint weighting to over constrained domains.** Our empirical work suggests constraint weighting does well for problems with distinctions between constraint groups. This led us to investigate solving real-world over constrained problems with hard and soft constraint groups and to introduce two dynamic constraint weighting heuristics that maintain a distinction between hard and soft constraint groups while still adding weights to violated constraints in a local minimum. In an empirical study, the dynamic schemes are shown to outperform other fixed weighting and non-weighting systems on a range of real world problems. In addition, the performance of weighting is shown to degrade less severely when soft constraints are added to the system, suggesting constraint weighting is especially applicable to realistic, hard and soft constraint problems.

Contents

1 Introduction **1**

1.1 Constraint Weighting for Constraint Satisfaction 1

1.1.1 Constraint Satisfaction 1

1.1.2 Constraint Satisfaction Algorithms 2

1.1.3 Constraint Weighting 3

1.2 Research Problems 4

1.3 Contributions 5

1.4 Outline 6

2 Constraint Satisfaction Techniques **7**

2.1 Definitions 7

2.2 Constructive Techniques 9

2.3 Local Search Techniques 11

2.3.1 Restart Strategies 15

2.3.1.1 Local Minimum Random Restart 15

2.3.1.2 Fixed Iteration Restart 15

2.3.1.3 GSAT 16

2.3.1.4 Value Propagation 17

2.3.2 Stochastic Strategies 18

2.3.2.1 Simulated Annealing 18

2.3.2.2 WSAT 20

2.3.3 Memory Strategies 21

2.3.3.1 Tabu Search 21

2.3.3.2 HSAT, NOVELTY and RNOVELTY 23

2.3.4 Weighting Strategies 25

2.3.4.1 Developments in Constraint Weighting 27

2.3.4.2 Constraint Weighting and Tabu Search 28

2.4 Summary 29

3 Modelling Realistic Problems 30
3.1 Specific and General Solutions 30
3.2 Problem Descriptions 31
3.3 Binary vs. Non-Binary Representation 33
3.3.1 Transforming Non-Binary CSPs 33
3.3.2 Domain Size Issues in Non-Binary Transformations 35
3.3.3 Partial Non-Binary to Binary Transformation 36
3.3.4 Defining Constraints for Tupled Domains 38
3.3.5 Lessons for General Problems 40
3.4 Representing Complex Move Operators 41
3.4.1 Making a Move in a Timetabling Problem 41
3.4.2 Defining Array-based Local Search Constraints 43
3.5 Summary 49

4 Constraint Weighting 51
4.1 Background and Motivations 51
4.2 Constraint Weighting Algorithms 53
4.3 WSAT and Tabu Search Algorithms 55
4.4 Experimental Results 56
4.4.1 Satisfiability Results 56
4.4.2 CSP Results 60
4.4.3 Constraint Weight Curves 62
4.4.4 Constraint Trajectories 66
4.4.5 Measuring Constancy 70
4.4.6 Measuring Problem Structure 71
4.5 Analysis 75
4.5.1 Constraint Weighting Behaviour 75
4.5.2 Identifying Hard Constraint Groups 76
4.5.3 Scaling Effects 80
4.5.4 Overall Behaviour 81
4.6 Summary 84

5 Improving Constraint Weighting 85
5.1 Background and Motivations 85
5.2 Hybrid Techniques 86
5.3 Arc Weighting 88
5.3.1 An Efficient Network Representation 89
5.3.2 Modifications to the Weighting Algorithm 91
5.4 Arc Weighting Experimental Results 92
5.4.1 Arc Weighting on Specialised and General Problem Domains 92
5.4.2 Arc Weighting Performance 94
5.5 Analysis of Arc Weighting 95
5.5.1 Distinguishing Moves 95
5.5.2 Arc Weighting Costs 96
5.5.3 Effects of Problem Size 98
5.5.4 Divergence 99
5.5.5 Applicability to Other Domains 99
5.6 Hybrid Experimental Results 100
5.7 Analysis of Hybrid Algorithm Performance 101
5.8 Summary 102

6 Over-Constrained Problems 104
6.1 Background and Motivations 104
6.2 Constraint Weighting for Over-Constrained Problems 106
6.2.1 Weighting with Hard and Soft Constraints 107
6.2.2 Dynamic Constraint Weighting 110
6.3 Experiments 112
6.3.1 Control Algorithms 112
6.3.2 Comparison Algorithms 113
6.3.3 Test Problems 114
6.3.4 Results 116
6.4 Analysis 117
6.4.1 Nurse Rostering 117
6.4.2 Timetabling 119
6.4.3 RLFAPs 120

6.4.4 Overall Comparison 122
6.5 Summary 126

7 Conclusion **127**
7.1 Summary 127
7.2 Future Work 130

Appendix: Zero One Block Constraints **133**

Bibliography **135**

List of Figures

1.1 An example solution to a four queens chess problem 2
1.2 Constraint weighting four queens example 3

2.1 A backtracking algorithm 9
2.2 Thrashing behaviour in backtracking 10
2.3 A general local search algorithm 12
2.4 Hill-climbing version of GenerateLocalMoves 13
2.5 Hill-climbing version of MakeLocalMove 14
2.6 Example local search topologies [Morris, 1993] 15
2.7 Graphical analysis of optimal restart value 16
2.8 GSAT version of GenerateLocalMoves 17
2.9 SA version of GenerateLocalMoves 18
2.10 SA version of MakeLocalMove 18
2.11 WSAT version of MakeLocalMove 19
2.12 WSAT version of GenerateLocalMoves 20
2.13 Tabu search version of GenerateLocalMoves. 21
2.14 RNOVELTY version of GenerateLocalMoves 24
2.15 RNOVELTY version of MakeLocalMove. 25
2.16 Using constraint weighting for graph colouring 25
2.17 Constraint weighting version of GenerateLocalMoves. 26

3.1 Non-binary and binary constraint graphs 34
3.2 A non-binary staff requirement constraint 35
3.3 Nurse domain values for simplified problem 38
3.4 Example solution for simplified problem 39
3.5 All purpose getCostChange method 44
3.6 testChange method for `alldifferent` constraint 44
3.7 Array version of testChange method for `alldifferent` 45

3.8 A violated `block` constraint 46
3.9 A satisfied `block` constraint 46
3.10 Array version of testChange method for `block` constraint 47
3.11 getBlockLength method for `block` constraint 47
3.12 An unsatisfied `gap` constraint 48
3.13 A satisfied `gap` constraint 48
3.14 getGapLength method for `gap` constraint 48
3.15 Array version of testChange method for `gap` constraint 49

4.1 Three strategies for constraint weighting 54
4.2 Result plot for large DIMACS 3-SAT problems 58
4.3 An example constraint weight curve 62
4.4 Constraint weight curves for various 3-SAT problems 63
4.5 Constraint weight curves for different constraint weight methods . . . 63
4.6 3-SAT and log function comparison 64
4.7 Non-uniform DIMACS constraint weight curves 64
4.8 Uniform DIMACS constraint weight curves 65
4.9 CSP constraint weight curves 65
4.10 Changing weight order for 4 selected constraints 67
4.11 Weight trajectories of the 24 most heavily weighted *AIM* 1 constraints . 68
4.12 Weight trajectories of the 2nd 10 most heavily weighted *AIM* 1 constraints 69
4.13 Weight trajectories of the first 17 most heavily weighted *r100* constraints 69
4.14 Constancy measure *Ct* of the top 10% of the heaviest weighted constraints 70
4.15 Neighbour count ranges as a proportion of random neighbour counts . . 74
4.16 Neighbour count std deviations as a proportion of random neighbour counts 74
4.17 Roster and timetabling constraint weight curves 78
4.18 Top 5% of roster and timetabling constraint weight curves 79

5.1 NOVELTYWGT version of GenerateLocalMoves 87
5.2 A simple constraint weighting scenario 88
5.3 Arc weight cost function 90
5.4 Arc weight version of GenerateLocalMoves 91
5.5 Proportion of solved problems by time 96

5.6 Proportion of solved problems by iterations 97
5.7 Comparison of hill climbing moves 97
5.8 Number of minima by iterations 98
5.9 Number of solved satisfiability problems by iterations 99

6.1 GenerateLocalMoves for over-constrained constraint weighting . . . 106
6.2 Weighting hard and soft constraints 109
6.3 The Flexible Weight Adjustment algorithm 112
6.4 Nurse rostering anytime curves for weighting algorithms 118
6.5 Nurse rostering anytime curves for comparative algorithms 119
6.6 Timetabling anytime curves for weighting algorithms 120
6.7 Timetabling anytime curves for comparative algorithms 121
6.8 RLFAP anytime curves for weighting algorithms 122
6.9 RLFAP anytime curves for comparative algorithms 122

List of Tables

2.1 A local search taxonomy 28

3.1 An example tupled nurse variable domain. 37

4.1 Results for small 3-SAT problems 57
4.2 Results for structured DIMACS problems. 59
4.3 Results for CSPs 61
4.4 Averaged small world measures for each problem set 72
4.5 Statistics for variable neighbour counts by problem domain 73
4.6 Parameter settings for WSAT and TABU algorithms 83

5.1 Comparison of mean performance values 94
5.2 Table 5.1 ARCWGT values as a proportion of MINWGT values . . 94
5.3 Comparison of iteration speed 98
5.4 3-SAT results for hybrid weighting algorithms 101
5.5 DIMACS results for hybrid weighting algorithms 101

6.1 Averaged results for 16 nurse rostering problems 117
6.2 Averaged results for 10 random timetabling problems. 120
6.3 Averaged results for 4 RLFAPs (1,2,3 and 11) 121
6.4 Comparison of Chapter 4 and Chapter 6 nurse rostering success rates 123
6.5 Comparison of Chapter 4 and Chapter 6 timetabling success rates . . 123
6.6 Comparison of original and adapted GLS performance 125

Definitions of Abbreviations and Terms

AIM refers to satisfiability problems created using an AIM generator (named after *A*sahiro, *I*wama and *M*iyano, see [Asahiro *et al*., 1993]). The special feature of an AIM generator is that it can build single solution problems.

ARCWGT a local search constraint weighting heuristic that additionally weights constraints that are simultaneously violated at a local minimum.

BEST a stochastic local search heuristic that either moves randomly or selects the best cost move according to a probability or noise level *p*.

BESTWGT a local search heuristic that combines BEST and MOVEWGT.

bin40 refers to the randomly generated *bin*ary CSPs used in the thesis with 30 variables, each with 10 domain values, a constraint density of *40*% and a constraint tighness of 32%.

bin80 refers to the randomly generated *bin*ary CSPs used in the thesis with 30 variables, each with 10 domain values, a constraint density of *80*% and a constraint tighness of 17%.

CNF *C*onjunctive *N*ormal *F*orm: CNF problems are made up of a conjunction of clauses of disjunct literals.

CSP *C*onstraint *S*atisfaction *P*roblem: a CSP is a problem expressed in terms of variables with domain values and constraints that define the allowable combinations of domain values for the variables. A solution to a CSP is an instantiation of all variables such that all the constraints are satisfied.

Ct Constancy measure that looks at the amount of change in the top 10% of weighted constraints during a search.

DIMACS benchmark refers to the set of benchmark satisfiability problems available from the Center for *DI*screte *MA*thematics and *C*omputer *S*cience at ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf.

DWA *D*ownward *W*eight *A*djustment: a dynamic local search constraint weighting heuristic for hard and soft constraint problems where the hard constraint weight multiplier is initially set to the total number of soft constraints + 1. Hard weight is then adjusted downwards during the search to equal the number of soft constraints violated in the best solution found so far + 1.

EFLOP *E*scaping *F*rom *L*ocal *O*ptima by *P*ropagation: a local search heuristic that uses value propoagation to escape from local minima [Yugami *et al*., 1994].

FWA *F*lexible *W*eight *A*djustment: a dynamic local search constraint weighting heuristic for hard and soft constraint problems where the hard constraint weight multiplier is initially set to the weight of a soft constraint + 1. This weight is then incremented at a local minimum if any hard constraints are violated, otherwise it is decremented.

GLS *G*uided *L*ocal *S*earch: a local search technique developed by [Voudouris and Tsang, 1996] that penalises problem features at a local minimum according to a utility function.

GSAT original local search heuristic proposed by Selman *et al*. [1992] for solving satisfiability problems.

HSAT a variant of GSAT proposed by [Gent and Walsh, 1993] that breaks ties on equal cost moves by considering when a move was last made.

***k*-SAT** refers to satisfiability problems with a fixed number of *k* literals in each clause, e.g. 3-SAT problems all have 3 literals per clause.

ii32 refers to the *i*nductive *i*nference problems from the DIMACS challenge set used in the thesis (namely ii32b3, ii32c3, ii32d3 and ii32e3).

MAX a local search constraint weighting heuristic for hard and soft constraint problems where the hard constraint weight multiplier is fixed to the total number of soft constraints + 1.

MAX-SAT refers to over-constrained satisfiability problems where the objective is to satisfy as many clauses as possible.

MIN a local search constraint weighting heuristic for hard and soft constraint problems where the hard constraint weight multiplier is fixed to the number of soft constraints violated in an optimal solution + 1.

MINWGT a local search constraint weighting heuristic that adds weight to violated constraints at a local minimum.

MOVEWGT a local search constraint weighting heuristic that adds weight to a violated constraint when an overall cost improving move that also improves the constraint cannot be found.

NOVELTY a stochastic local search heuristic proposed in [McAllester *et al*., 1997] that evaluates moves based on how recently the move was last made.

NOVELTYWGT a local search heuristic that combines NOVELTY and MOVEWGT.

par refers to the *par*ity function learning problems from the DIMACS challenge set used in the thesis (namely par8-2-c and par8-4-c).

PCSP *P*artial *C*onstraint *S*atisfaction *P*roblem: a formalism for representing and solving over-constrained problems by searching for a solution that partially satisfies the problem constraints (from [Freuder and Wallace, 1992]).

r100 (also *r200* and *r400*) refers to randomly generated 3-SAT problems (see *k*-SAT above) with a clause to variable ratio in the cross-over region of 4.3 : 1. *r100* refers to 100 variable problems, *r200* to 200 variable problems, etc.

RLFAP refers to *R*adio *L*ink *F*requency *A*ssignment *P*roblems based on the real problem of assigning frequencies to radio links (made available by the French Centre d'Electronique l'Armament at listserver@saturne.cert.fr).

RNOVELTY a stochastic local search heuristic proposed in [McAllester *et al*., 1997] that evaluates moves based on how recently the move was last made and the relative costs of the two most promising moves.

RNOVELTYWGT a local search heuristic that combines RNOVELTY and MOVEWGT.

SA *S*imulated *A*nnealing: a stochastic local search heuristic modelled after the physical cooling process of heated atoms [Abramson, 1992].

ssa refers to the circuit fault diagnosis problems from the DIMACS challenge set used in the thesis (namely ssa7552-038, ssa7552-158, ssa7552-159 and ssa7552-160).

TABU a constraint sampling local search heuristic that avoids undoing recently made moves [Glover, 1989].

TABUWGT a local search heuristic that combines TABU and MOVEWGT.

tt_rand refers to the randomly generated timetabling problem set used in the thesis where classes are assigned student groups, staff and room requirements on a random basis.

tt_struct refers to the randomly generated timetabling problem set used in the thesis that reflects the structure of a realistic problem.

UTILWGT a constraint weighting algorithm based on the utility function proposed in [Voudouris and Tsang, 1996].

WSAT refers to a family of local search techniques that grew out of the original WalkSat heuristic [Selman *et al*., 1994] (includes BEST, NOVELTY and RNOVELTY).

Acknowledgments

I would like to thank my supervisor Dr. Abdul Sattar for his tireless support and encouragement, and for always pointing me in the right direction. I would also like to thank Dr. Clyde Wild and the Gold Coast Campus of Griffith University for their generous financial assistance. Finally, I would like to thank Byungki Cha, Paul Morris, Bart Selman, Peter van Beek and Benjamin Wah for sharing their code, helpful comments and correspondence during the process of completing this thesis.

Statement of Originality

This work has not previously been submitted for a degree or diploma to any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Signed: ----------

January 2000

Chapter 1

Introduction

In this chapter we informally introduce the idea of applying constraint weighting to constraint satisfaction. We then describe the problems addressed in the thesis and explain our motivation for solving them. Finally we present a summary of the contributions of the thesis and an outline of the remaining chapters.

1.1 Constraint Weighting for Constraint Satisfaction

1.1.1 Constraint Satisfaction

Representing and solving problems involving constraints has important applications in artificial intelligence, including satisfiability testing, scheduling, image interpretation and planning. The idea of constraint satisfaction is to represent problem knowledge by defining constraints on the allowable values of problem variables. In this way we can model many different problems within a common framework and so develop algorithms that exploit this framework (rather than concentrating on solving individual problems).

As an example, consider the well-known n-queens problem: here the aim is to place n queens on an $n \times n$ chessboard so that no two queens are attacking one another. To transform this into a constraint satisfaction problem (CSP) we need to identify the *variables* in the problem (the things that can change, i.e. the queens), the *domains* of the variables (the values that each variable can assume, i.e. the chessboard squares) and the *constraints* between the variables (i.e. the limitation that no two queens can be on squares that are in the same row, column or diagonal). A CSP is

solved by finding an answer where all variables are instantiated (i.e. all queens are on the board) and all constraints are satisfied (i.e. no queen is attacking another). An example solution to the four queens problem is shown in figure 1.1.

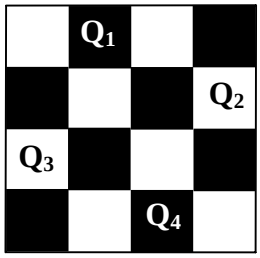


Fig. 1.1. An example solution to a four queens chess problem

1.1.2 Constraint Satisfaction Algorithms

Constraint satisfaction algorithms can be placed in two general categories:

- Constructive (backtracking) algorithms
- Local search (iterative repair) algorithms

A constructive algorithm builds up answers incrementally, checking at each stage that all constraints are satisfied. In the *n*-queens example, this means placing queens on the board one at a time, making sure each new queen is not attacked by a previous queen, until an answer is found or there are no more unattacked squares. If no unattacked squares are available, the algorithm will *backtrack*, undo an earlier move, and continue on again. Such algorithms are *systematic* (i.e. they are guaranteed to find all possible solutions to a problem) but on many problems have worst case exponential time complexity [Mitchell 1998].

Local search techniques prove useful as problem sizes grow and the performance of constructive techniques starts to decline. Although not guaranteed to find an answer, and with unpredictable performance, local search techniques have proved the best practical alternative for many larger CSPs. A local search strategy starts with a complete but flawed answer to a problem and then tries to find ‘local’ moves that improve the overall cost of the answer. For the *n*-queens problem, this means starting with all *n* queens on the board and searching the domain of each queen for moves that reduce the total number of attacks. As each move changes the situation for all the other queens, we can repeatedly find the best move for different queens until a solu-

tion is found or there are no moves left that reduce the number of attacks. In this second case we have reached a *local minimum*. The challenge for all non-trivial local search techniques is to find the best way to avoid or escape local minima and carry on the search.

1.1.3 Constraint Weighting

In the early 90's, [Morris, 1993] proposed a new local search heuristic for satisfiability testing called Breakout. At the same time, [Selman and Kautz, 1993] proposed a similar clause weighting algorithm and later [Thornton and Sattar, 1997] introduced a constraint weighting heuristic for solving general CSPs. All three techniques share the same basic mechanism for escaping or avoiding local minima: placing weights on unsatisfied constraints. This makes answers that violate weighted constraints more costly, changing the structure of the problem so that other answers become more attractive. For example, consider the situation in figure 1.2a: Q_1 is attacking Q_2, and there is no single move that can improve the situation. Constraint weighting would increase the cost of violating the diagonal constraint between Q_1 and Q_2, making any position that violates another constraint more attractive. Hence we can move Q_2 to the position in figure 1.2b and from there we can move Q_4 and arrive at the solution in figure 1.1.

(a) (b)

Fig. 1.2. Constraint weighting four queens example

Constraint weighting techniques have proved effective on smaller hard satisfiability problems [Morris, 1993; Cha and Iwama, 1995], leading to the development of specialised algorithms for satisfiability [Cha and Iwama, 1996; Castell and Cayrol, 1997; Frank, 1996] and the application of weighting to over-constrained problems [Cha *et al.*, 1997], scheduling [Thornton and Sattar, 1997], neural networks [Davenport *et al.*,

1994] and genetic algorithms [Bowen and Dozier, 1996]. In addition, other related techniques such as Guided Local Search (GLS [Voudouris and Tsang, 1996]) and the discrete Lagrangian method [Wu and Wah, 1999] use the principle of constraint weighting but explain and apply it in different ways. *Basic* constraint weighting (as proposed by Morris [1993]) does not require the tuning of parameter values to obtain optimum performance and has no domain dependent features[1]. This makes it an excellent candidate as a general purpose constraint solving algorithm. As products like ILOG® have shown, there is significant scope for the practical application of constraint technology, and the techniques developed here are directly relevant to solving larger and/or over-constrained problems within such general purpose systems.

1.2 Research Problems

As we have seen, the constraint satisfaction paradigm models the world using variables, domains and constraints. In many practical applications, standard CSP representations, while capturing the essential problem features, produce models that cannot be solved efficiently using general purpose CSP algorithms. This typically leads to the development of problem specific techniques and the abandonment of a general approach. We address this area by looking at the application of CSP techniques to two complex, real world scheduling problems, and examine general extensions to the CSP framework that can be used to produce efficiently solvable models.

Constraint weighting was originally developed as a method for solving satisfiability problems. Outside of the satisfiability domain, the relative performance of constraint weighting in comparison with other local search techniques is poorly understood. We address this in an empirical study which looks at techniques from satisfiability [McAllester *et al.*, 1997] and tabu search [Glover, 1989], and investigates algorithm performance on a range of different CSPs. In the process, we examine the behaviour of several constraint weighting schemes and look for problem types for which weighting is more applicable.

Next, we look at the problem of improving the performance of constraint weighting. Existing enhancements have concentrated on satisfiability testing and produced heuristics that are not applicable to the broader domain of constraint satisfaction [Cha

[1] this is not true for all weighting techniques, for instance GLS does use parameters

and Iwama, 1996; Castel and Cayroll, 1997]. To address this, we propose a domain independent arc weighting algorithm that weights binary connections between constraints that are simultaneously violated in a local minimum. In a second empirical study we compare the performance of arc weighting with a standard constraint weighting technique introduced earlier in the thesis. In addition we consider several hybrid algorithms that introduce a weighting component into existing non-weighting methods and empirically evaluate the benefits of mixing these techniques.

Finally, we extend the application of constraint weighting to over-constrained problem domains containing hard (mandatory) and soft (desirable) constraints. This work is motivated by the common appearance of hard and soft constraints in realistic problems and the lack of a constraint weighting heuristic that can maintain the long-term distinction between hard and soft constraints. We propose two dynamic constraint weighting schemes and evaluate their performance in comparison with two fixed weighting schemes, and four other non-weighting algorithms.

1.3 Contributions

The main contributions of the thesis are:

- The extension and practical application of the CSP modelling framework to include array-based domains and array-based domain use constraints.
- The characterisation of the behaviour of constraint weighting using constraint weight curves and measures of weighting constancy and problem structure.
- The recognition that constraint weighting is best suited to problems where there is a clear distinction between a difficult constraint group and the remaining easier constraints.
- The development and evaluation of a range of pure and hybrid constraint weighting schemes.
- The development of an efficient arc weighting algorithm that is shown to outperform standard constraint weighting within a general constraint solving system.
- The development of two dynamic constraint weighting heuristics that can solve problems involving hard and soft constraints and that outperform other fixed weighting and non-weighting systems.

1.4 Outline

In the next Chapter we give a general survey of constraint satisfaction techniques, concentrating on local search and the specific algorithms used in the remainder of the thesis. Then, in Chapter 3, we examine the modelling issues involved in efficiently solving two scheduling problems using CSP techniques. As a result of this, we propose several extensions to the standard CSP representation. In Chapter 4, we present an empirical study comparing constraint weighting with several recently proposed satisfiability techniques and with an implementation of tabu search. We also evaluate three versions of constraint weighting: move-based, local minimum-based and utility-based. As part of this study, we graphically analyse the behaviour of constraint weighting and look for connections between algorithm performance, weighting behaviour and problem structure. In Chapter 5, we propose a new domain independent arc weighting algorithm that uses information about the frequency that constraints are simultaneously violated. We present an empirical study of arc weighting in comparison with a standard weighting scheme using problem domains introduced in Chapter 4. In addition we further experiment with adding weighting schemes into other algorithms. In Chapter 6 we propose two dynamic constraint weighting schemes for solving over-constrained problems involving hard and soft constraints. The schemes are evaluated on a range of over-constrained problems adapted from domains introduced earlier in the thesis, and in comparison to two fixed weighting schemes and an alternative dynamic weighting scheme for tabu search. Finally, in Chapter 7, the overall results and conclusions of the thesis are summarised and avenues for future work are presented.

Chapter 2

Constraint Satisfaction Techniques

In this chapter we review the areas of constraint satisfaction of relevance to the thesis. We start with some formal definitions and a brief outline of the constructive approach to constraint satisfaction. We then present a taxonomy of local search techniques based on the method used to escape or avoid a local minimum solution.

2.1 Definitions

Constraint Satisfaction Problem (CSP): Formally, a constraint satisfaction problem (CSP) consists of a set V of n variables, $\{v_1, v_2, ..., v_n\}$, with each v_i having a domain D_i of possible values. Constraint relations are defined on subsets of V, and consist of subsets of the Cartesian products of the domains of the variables that participate in the constraint. Each constraint relation tuple represents a combination of variable values from the subset of variables over which the constraint is defined, that satisfy the particular conditions of that constraint. Solving a CSP involves finding an n-tuple of values for each variable in V such that all constraint relations are satisfied. A CSP may require the complete set of n-tuple solutions, one member of the set, or to discover whether the set has any members [Mackworth, 1977].

Partial Constraint Satisfaction Problem (PCSP): The above CSP definition covers problems where *all* constraints must be satisfied for an answer to exist. Many realistic problems are over-constrained, meaning there is no answer that satisfies all constraints. In this case we can model the problem as a Partial Constraint Satisfaction Problem (PCSP). A PCSP is here defined as a CSP, P, with the addition definition of

a solution space *S*, a cost function f and a maximum solution cost *C* [Freuder and Wallace, 1992]. *S* is the set of all possible *n*-tuples of values for each variable in *V* in *P* and f measures the distance between elements of *S* in terms of the number and importance of the constraints violated. By relaxing the constraints of the original problem *P*, we can visit each solution $s \in S$. If a solution is found such that $f(s) \leq C$, then a solution to the PCSP is also found (we assume if $f(s) > 0$ then *s* violates at least one constraint, hence a CSP can be defined as a PCSP where $C = 0$).

Local Search Terminology: Following on from the definition of a PCSP, we can define a local search space $LS \subseteq S$, where *LS* is the set of all solutions that can be reached from some initial point $s_0 \in S$ by recursively applying a local neighbourhood function *N*. *N* generates and applies a set of moves *M*, such that each solution $s' \in N(s)$ is exactly one move $m \in M$ away from *s* (each s' is therefore called a *neighbour* of *s*). A local search moves between successive neighbouring solutions $s_0, s_1, .. s_n$ in *LS*, by selecting a move *m* from *M*, denoted by $s_{i+1} \leftarrow s_i \oplus m$ (see Section 2.3).

Conjunctive Normal Form (CNF) Problem: Many of the recent advances in local search techniques have occurred in solving Boolean Satisfiability problems in conjunctive-normal form (CNF). A CNF formula consists of a conjunction of clauses, where each clause is a disjunction of literals and each literal is a propositional variable or its negation [Poole *et al*., 1998]. For example, consider a CNF formula with three propositional variables, *x*, *y* and *z*, and the following four clauses:

$$\underset{(1)}{\{\neg x \vee \neg y \vee \neg z\}} \wedge \underset{(2)}{\{x \vee y \vee \neg z\}} \wedge \underset{(3)}{\{\neg x \vee y\}} \wedge \underset{(4)}{\{\neg y \vee z\}}$$

The formula is satisfiable if values for *x*, *y* and *z* exist, such that all clauses simultaneously evaluate to true. For example, *x* = *false* satisfies clauses 1 and 3, *y* = *true* satisfies clauses 2 and 3, and z = *true* satisfies clause 4. In this case all four clauses are true and the formula is satisfied. A CNF satisfiability problem can be easily formulated as a non-binary CSP by taking each clause as a constraint and each literal as a variable with two domain values: {*true, false*}. Alternative binary CSP encodings are

possible using the dual and hidden variable techniques explained in Section 3.3.1 (for more detail see [Walsh, 2000]).

2.2 Constructive Techniques

Constructive techniques try to build consistent solutions incrementally. In solving CSPs, this has meant the use of backtracking, consistency techniques and structure-driven algorithms [Kumar, 1992]. Analogous constructive techniques have also been applied to PCSPs using branch and bound [Freuder and Wallace, 1992].

The constructive technique most relevant to the thesis is the backtracking algorithm (see figure 2.1). Backtracking starts by selecting a value for an initial variable and then tries to extend the solution by selecting a value for a second variable, such that the two values are consistent (i.e. there are no constraint violations). This partial solution is then extended to a third variable, and so on, until either all variables are instantiated with consistent values, or a variable is found with no remaining consistent domain value. In this case (known as a dead-end), the algorithm will return (or backtrack) to a previously instantiated variable and try another value. If no other consistent value can be found for this variable, the algorithm will backtrack further until a new consistent value is found for some other variable or no more consistent values are available.

```
procedure Backtrack(V_left, V_done, S)
begin
   if V_left ≠ ∅ then
   begin
      v_i ← SelectVariable(V_left)
      for each d_ij ∈ D_i of v_i do
      begin
         v_i ← d_ij
         if not InConflict(v_i, V_done) then Backtrack(V_left - v_i, V_done ∪ v_i)
      end
   end
   else if S = ∅ then S ← V_done
end
begin program
   S = ∅,   Backtrack(V, ∅, S)
end program
```

Fig. 2.1. A backtracking algorithm

Simple backtracking is guaranteed to find all consistent solutions to a CSP, as it visits all consistent instantiations of a given variable ordering. In addition, backtracking prunes the search space by ignoring areas of the search tree that exist beyond a dead-end, and so is more efficient than a simple exhaustive search. However, as the general task of solving a CSP is NP-complete [Mitchell, 1998], such pruning cannot be guaranteed to produce a polynomial time algorithm. The main problem for backtracking is *thrashing* [Mackworth, 1987]. Thrashing refers to repeated failure at dead-ends for the same underlying reasons. For example, figure 2.2 shows part of the search tree for a three variable (v_1, v_2, v_3), two domain value (d_{i1}, d_{i2}) CSP. If there is no value for v_3 that is consistent with d_{11} for v_1, we will repeatedly rediscover this conflict at different parts of the tree and consequently fail for the same underlying reason:

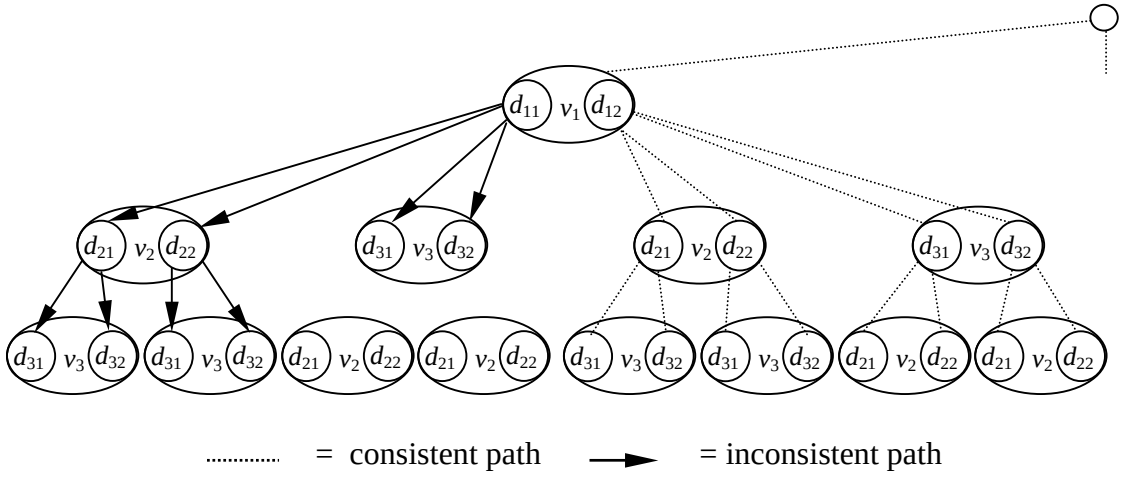


Fig. 2.2. Thrashing behaviour in backtracking

Three main strategies have been developed to improve the efficiency of backtracking:

1. **Consistency-enforcing algorithms**: Consistency-enforcing algorithms (e.g. arc-consistency, path-consistency, etc [Mackworth, 1987]) recognise domain values that cannot be a part of any complete solution. These values are then either deleted from their domains or new constraints are inferred that forbid the inconsistent value combinations (for example in figure 2.2 we could delete d_{11}).

2. **'Intelligent' backtracking techniques**: Intelligent backtracking techniques are generally divided into look-back and look-ahead schemes. Look-back schemes avoid unnecessary work by learning from already instantiated variables. For ex-

ample, backmarking [Gaschnig, 1977] avoids repeating previous consistency checks, while backjumping [Gaschnig, 1978] avoids backtracking to variables that are not currently in conflict. Look-ahead schemes examine the effects of current moves on future uninstantiated variables, so that potential dead-ends can be detected earlier (e.g. forward-checking [Haralick and Elliott, 1980] and maintaining arc-consistency [Sabin and Freuder, 1997]).

3. **Variable and value ordering heuristics**: Variable ordering heuristics generally use a 'fail-first' principle to reduce the size of the search tree as fast as possible. This can involve dynamically selecting the variable with the fewest remaining domain values [Bitner and Reingold, 1975] or selecting the variable involved in the largest number of constraints [Dechter, 1992]. In addition, value ordering heuristics can help to avoid dead-ends, for instance, by selecting values that least reduce the number of values available for future variables [Dechter, 1992].

2.3 Local Search Techniques

On many CSPs, constructive algorithms have exponential time complexity (e.g. all known NP-complete problems). In these situations, as problem sizes get larger, alternative non-systematic local search techniques become more practical. As introduced in Chapter 1, a local search starts with a complete, but inconsistent solution, and then attempts to iteratively improve or repair constraint violations. Because a local search can move in the space of inconsistent solutions, it can be used to solve both CSPs and PCSPs without significant modification.

Many well-known algorithms can be classified as local search techniques, ranging from 'greedy' hill-climbing algorithms to more specific approaches such as the simplex algorithm in linear programming [Dantzig, 1963]. The connecting principle is that all the techniques search for improving solutions in the local neighbourhood of an existing solution. Generally if an improving solution is found then the search tries to find an improving solution for the new solution, otherwise behaviour depends on the particular technique that is employed [Papadimitriou and Steiglitz, 1982]. As introduced in Section 2.1, a solution s_{i+1} in the neighbourhood of an existing solution s_i is created by selecting a move $m \in M$, generated by a neighbourhood function N. Moves

can simply change the domain value for one variable, or can change several variables and include heuristics that ensure the generated move satisfies certain constraints. For instance, in the travelling salesman problem, 2-OPT moves delete two non-adjacent edges of a tour and then add back the unique two edges that create a new tour [Glover, 1989]. In addition, various move selection heuristics have been proposed, some accepting equal cost moves (e.g. GSAT, [Selman *et al*., 1992]) and others accepting cost increasing moves (e.g. Simulated Annealing, [Abramson, 1992]). Nevertheless, all techniques share the same basic approach. This is shown in figure 2.3, which uses the notation introduced in Section 2.1, where s is the current solution represented as variable value pairs (v_i, d_{ij}), such that $s \in LS$, $v_i \in V$, $d_{ij} \in D_i$ and $M' \subseteq M$, where M is the set of all local moves for the solution s (for generality the exact procedure for selecting the initial (v_i, d_{ij}) pairs in s is left undefined).

```
procedure LocalSearch(MaxCost, MaxMoves)
begin
   for each v_i ∈ V do s ← s ∪ {(v_i, d_ij) | d_ij ∈ D_i}
   while f(s) > MaxCost and TotalMoves < MaxMoves do
   begin
      M' ← GenerateLocalMoves(s, TotalMoves)
      if M' ≠ ∅ then MakeLocalMove(s, M', TotalMoves)
   end
end
```

Fig. 2.3. A general local search algorithm

Within the structure of figure 2.3 we can characterise different local search strategies by redefining the GenerateLocalMove and MakeLocalMove functions. Firstly, figures 2.4 and 2.5 define GenerateLocalMoves and MakeLocalMove for a hill-climbing local search. The GenerateLocalMoves procedure returns the set of all best cost moves M' in the neighbourhood of s, where a move $m \leftarrow \{v_i, d\}$ consists of instantiating variable v_i with domain value d.

The hill-climbing local search algorithm is the basis of all local search techniques. Although simple, it has proved robust and effective at solving a wide range of CSPs and PCSPs [Minton *et al*., 1992; Glover, 1989; Thornton, 1995]. In comparison to a constructive approach, local search looks at a complete rather than a partial instantiation of variables, and so knows the exact (rather than probable) cost of a move. For this reason it can move quickly to a low cost solution [Minton *et al*., 1992]. Unlike a

constructive algorithm, local search does not *systematically* cover the search space. Given a move operator that can connect all solutions and a mechanism to avoid running out of moves, a local search will eventually visit all solutions in the search space [Morris, 1993]. However, this involves the assumption of infinite time and the probability that the search will frequently revisit many of the same solutions. Consequently, worst case local search performance will usually be inferior to a constructive method, making local search impractical for finding a complete enumeration of acceptable cost solutions or for discovering that a problem cannot be solved.

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
   M' ← ∅, BestCost ← f(s) - δ      /* best cost slightly less than current cost */
   for each v_i ∈ V do if v_i in constraint violation then
   begin              /* only variables in violation can be in a cost reducing move */
      d_curr ← current domain value of v_i
      for each d ∈ D_i | d ≠ d_curr do              /* ignore current domain value */
      begin
         m ← {v_i, d}
         if f(s ⊕ m) ≤ BestCost then
         begin
            if f(s ⊕ m) < BestCost then
            begin
               BestCost ← f(s ⊕ m)
               M' ← ∅                          /* new best move so start again */
            end
            M' ← M' ∪ m                        /* move accepted as candidate */
         end
      end
   end
   if M' = ∅ then TotalMoves ← MaxMoves        /* local minimum so quit */
   return M'
end
```

Fig. 2.4. Hill-climbing version of GenerateLocalMoves

The idea of local search is to find a *short-cut* to an answer by descending quickly to the nearest minimum cost solution in the search space. It avoids the expense of a systematic search by exploiting the cost topography of the search space. The average case performance of a local search therefore depends on the particular cost surface of the problem being solved. For this reason local search techniques are usually evaluated empirically on a problem by problem basis rather than using formal analysis

techniques (although certain smaller problems have proved amenable to analysis, e.g. [Papadimitriou, 1994]).

procedure MakeLocalMove(*s, M′ , TotalMoves*)
begin
  randomly select *m* from *M′*
  $s \leftarrow s \oplus m$, $TotalMoves \leftarrow TotalMoves + 1$
end

Fig. 2.5. Hill-climbing version of MakeLocalMove

The main problem with a hill-climbing local search (i.e. one that only accepts cost improving moves) is that it descends to the *nearest* minimum cost solution in the search space. If no *single* move can improve on a solution, the search becomes stuck, even though it may not have found the global minimum. Although for certain problems (e.g. *n*-queens) a simple local search can be effective, most interesting problems have a search topography that contains many non-optimal local minima. One way to move on from a local minimum is to combine local search with a constructive approach and *backtrack* through the space of possible cost reducing moves [Minton *et al*., 1992]. However, several more sophisticated and powerful local search heuristics have been developed that escape or avoid local minima. These heuristics can be divided into four areas (according to the method of escape) forming the basis of the local search taxonomy used in the remainder of the thesis:

1. **Restart strategies** that restart the search either at a local minimum, or after a certain number of moves.
2. **Stochastic strategies** that allow cost increasing moves (according to a fixed or dynamically adjusted probability).
3. **Memory strategies** that remember previous moves or solutions and so avoid moves that lead back to an already visited solution.
4. **Constraint weighting strategies** that change the cost topography by dynamically adjusting the cost of violating selected constraints.

2.3.1 Restart Strategies

2.3.1.1 Local Minimum Random Restart

The simplest restart strategy is to randomly reassign all variable values each time a local minimum is encountered. By starting the search in a different area, we are likely to find a different local minimum (depending on the topography) and eventually to find a global minimum. However, this approach discards any information we could have learned in a previous search. In cases where a global minimum shares many common values with other local minima (as in figure 2.6a), restarting means we will have to relearn these assignments. In contrast, if the search space contains many evenly distributed local minima and there is no gradient towards a global solution (as in figure 2.6b) then there is little to learn from each minima and a random restart strategy becomes more efficient (note that in this situation a systematic search technique also becomes more competitive).

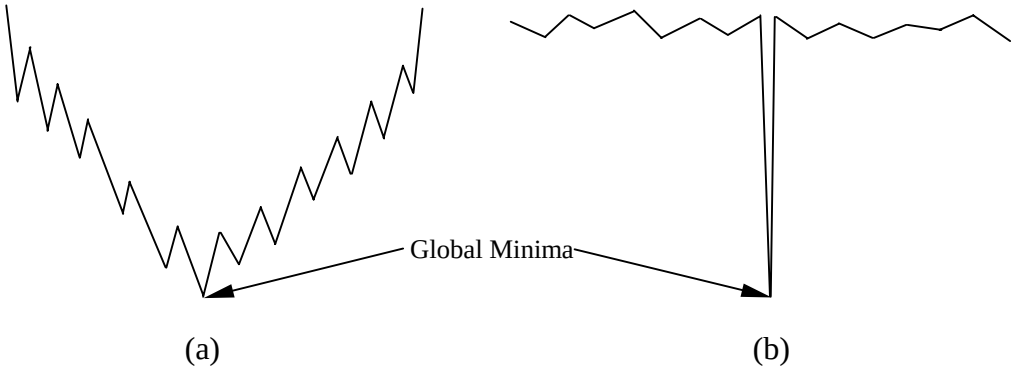
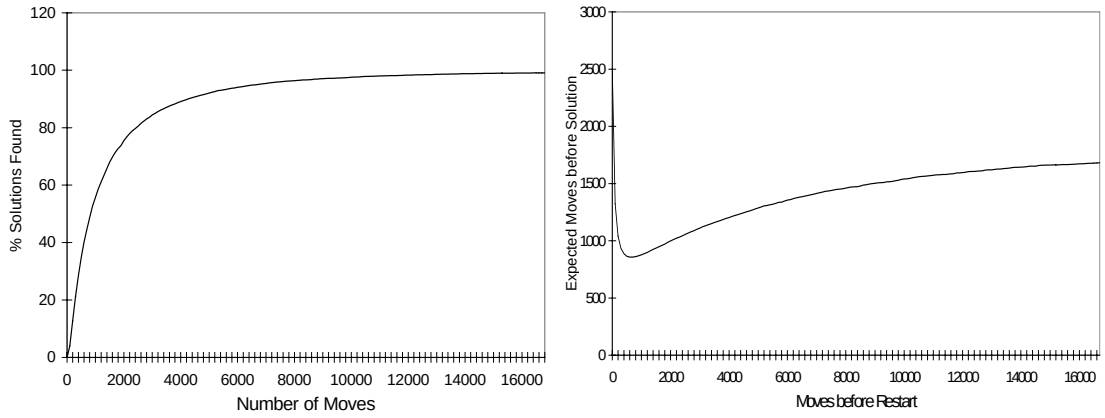


Fig. 2.6. Example local search topologies [Morris, 1993]

2.3.1.2 Fixed Iteration Restart

Fixed iteration restart restarts a problem after a fixed number of moves. The idea exploits the wide variation in the number of moves observed for local search techniques solving the same (usually hard) problem instances. For example, figure 2.7a plots the percentage of 10,000 runs on the same CNF satisfiability problem that are solved at different numbers of moves. Figure 2.7b uses this information to calculate the optimum number of restarts for the problem. The graphs show (for this problem and algorithm) it is optimal to restart an unsuccessful search after approximately 700 moves

rather than risk a slow search in the tail area. Empirical tests with various hard CNF satisfiability problems using GSAT have shown that the optimum restart point is fairly constant for a given problem type and size [Selman and Kautz, 1993].



(a) solution level by number of moves (b) using expected moves to estimate optimum

Fig.2.7. Graphical analysis of optimal restart value

2.3.1.3 GSAT

GSAT [Selman *et al.*, 1992] is a fixed iteration restart local search heuristic specifically developed for solving CNF satisfiability problems. The algorithm tries 'flipping' variables in the problem and accepts the best move that does not increase the number of unsatisfied clauses (breaking ties randomly). Flipping is equivalent to a move which tries each non-instantiated domain value for each variable, however, as SAT problem variables only have two values (true or false), a domain test simply changes a value from true to false or vice versa. Because GSAT accepts 'sideways' or equal cost moves the 'plateau' around a local minimum can be explored for another cost reducing move. If no cost reducing move exists on a plateau then different equal cost moves will be accepted indefinitely. For this reason, and to avoid the tail area (discussed in the previous section), the algorithm is artificially terminated after a certain number of moves (flips) and then restarted. The CSP version of GSAT (shown in figure 2.8) differs from the hill-climbing approach of Figure 2.4 in three areas:

1. All variables are considered, rather than just those in a constraint violation.
2. Equal or increasing cost moves are allowed
3. Hence GSAT does not recognise local minima (i.e. M' cannot $= \varnothing$).

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
   M′ ← ∅, BestCost ← ∞
   for each v_i ∈ V do
   begin
      d_curr ← current domain value of v_i
      for each d ∈ D_i | d ≠ d_curr do
      begin
         m ← {v_i, d}
         if f(s ⊕ m) ≤ BestCost then
         begin
            if f(s ⊕ m) < BestCost then
            begin
               BestCost ← f(s ⊕ m),   M′ ← ∅
            end
            M′ ← M′ ∪ m
         end
      end
   end
   return M′
end
```

Fig. 2.8. GSAT version of GenerateLocalMoves

Averaging-In: To avoid discarding information learnt in earlier searches, an averaging-in heuristic was proposed for GSAT [Selman and Kautz, 1993]. This heuristic records the initial and best solutions for the first search (i_1 and b_1), and generates the initial state for the second search (i_2) leaving variables that have the same assignment in i_1 and b_1 unchanged, and randomly assigning values to the variables that differ. From then on the initial state is generated by averaging in the two best previous solutions.

2.3.1.4 Value Propagation

Following on from Averaging-In, Yugami *et al*. [1994] proposed a more sophisticated restart strategy that involves moving from a local minimum using value propagation (called *E*scaping *F*rom *L*ocal *O*ptima by *P*ropagation or EFLOP). This method involves perturbing a local minimum by randomly selecting a variable in conflict, and changing its value. EFLOP then propagates this change by moving into a loop that tries to build a consistent sub-problem. This is done by selecting variables in newly unsatisfied constraints that a) have not been selected before in the current call of

EFLOP procedure b) were consistent in the original local minimum c) have a value that satisfies the constraint and d) are consistent with all other variables changed in the current EFLOP call. When no more variables meet these conditions EFLOP terminates and the local search is restarted.

2.3.2 Stochastic Strategies

Stochastic local search techniques escape or avoid local minima by adding a random element to the move selection heuristic that allows cost increasing moves. This is the basis of several techniques including simulated annealing (SA) and WSAT. (note GSAT and hill-climbing also use randomised selection to break ties and set up initial solutions but *not* as a method to escape local minima).

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
   if TotalMoves = 0 then
   begin
      M' ← ∅,
      for each v_i ∈ V do for each d ∈ D_i do M' ← M' ∪ {v_i, d}
   end
   return M'  /* i.e. return entire local neighbourhood */
end
```

Fig. 2.9. SA version of GenerateLocalMoves

```
procedure MakeLocalMove(s, M' , TotalMoves)
begin
   randomly select m from M'
   ΔE ← f(s ⊕ m) - f(s)
   if TotalMoves = 0 then T ← T_start
   else T ← T * R
   if ΔE < 0 or e^(-ΔE/T) > (random value between 0 and 1) then s ← s ⊕ m
   TotalMoves ← TotalMoves + 1
end
```

Fig. 2.10. SA version of MakeLocalMove

2.3.2.1 Simulated Annealing

Simulated annealing is a general purpose optimisation technique modelled after the physical cooling process of heated atoms [Abramson, 1992]. As with all local search,

a cost function is defined and local or neighbourhood solutions are generated according to a move operator. These solutions are automatically accepted if they produce a reduction in cost, but if a solution causes an increase in cost (also known as energy), it is accepted or rejected on the basis of an annealing probability function and the current system temperature [Connolly, 1992]. As the algorithm executes, the temperature of the system reduces (according to a cooling function), causing the probability of accepting an increased cost solution to reduce.

Various annealing probability and cooling functions have been proposed. As an example, classical annealing [Lo and Bavarian, 1992] uses a version of the Boltzman distribution to generate the probability of acceptance:

$$P(accept) = e^{-\Delta E/T}$$

where T = temperature and ΔE = change in cost caused by accepting the new solution. The temperature is then reduced using a geometric cooling schedule:

$$T_n = T_{n-1} * R$$

where R is the cooling rate $0 \leq R \leq 1$ and T is a positive real number [Abramson, 1992]. An example CSP simulated annealing algorithm is shown in figures 2.9 and 2.10. Here we allow the local neighbourhood to include all possible moves, rather than restricting selection to variables that are in conflict (as in hill-climbing). This is in line with the standard SA approach, although empirical tests on various CSPs suggest SA does better with a more selective choice of moves [Selman and Kautz, 1993; Thornton, 1995].

procedure MakeLocalMove(*s*, *M′* , *TotalMoves*)
begin
  randomly select *m* from *M′*
  if *f*(*s* ⊕ *m*) ≤ *f*(*s*) or *p* > (random number between 0 and 1) **then** *s* ← *s* ⊕ *m*
  TotalMoves ← *TotalMoves* + 1
end

Fig. 2.11. WSAT version of MakeLocalMove

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
   M′ ← ∅, BestCost ← ∞
   randomly select a violated constraint c
   if p > (random number between 0 and 1) then while M′ = ∅ do
   begin
      randomly select move m from domain of variables in c
      if m improves c then M′ ← M′ ∪ m
   end
   else for each v_next ∈ c do
   begin
      d_curr ← current domain value of v_next
      for each d ∈ D_next | d ≠ d_curr do
      begin
         m ← {v_next, d}
         if f(s ⊕ m) ≤ BestCost and m improves c then
         begin
            if f(s ⊕ m) < BestCost then
            begin
               BestCost ← f(s ⊕ m)
               M′ ← ∅
            end
            M′ ← M′ ∪ m
         end
      end
   end
   return M′
end
```

Fig. 2.12. WSAT version of GenerateLocalMoves

2.3.2.2 WSAT

WSAT [Selman *et al.*, 1994] is an extension of GSAT, again specifically developed to solve satisfiability problems. The version of WSAT we consider here is available from http://www.research.att.com/~kautz/walksat/ and differs from GSAT in restricting the move neighbourhood by randomly selecting a constraint in violation and then only considering the domain values of those variables in the constraint that cause the constraint to be satisfied (or improved). Then with probability *p*, a variable is selected at random from the constraint and its value is flipped, otherwise the best cost move is selected from the domain of the constraint. This allows the acceptance of cost increasing moves based on a probability threshold and so is similar to simulated annealing. However, in WSAT only variables involved in constraint violations are considered for

flipping, and the value of p is fixed during the search (i.e. is not sensitive to the size of cost increase and does not decay over time). Additionally the WSAT cost function selects moves on the basis of minimising the number of constraints a move will violate, ignoring the constraints that become satisfied. A version of WSAT for solving general CSPs is shown in figures 2.11 and 2.12 (more recently developed WSAT heuristics are discussed in section 2.3.3.2).

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
   M' ← ∅
   BestCost ← ∞     /* setting best cost to ∞ allows cost increasing moves */
   for each v_i ∈ V do
   begin
      d_curr ← current domain value of v_i
      for each d ∈ D_i | d ≠ d_curr do
      begin
         m ← {v_i, d}
         if TotalMoves - LastUse(m) ≤ MaxListLength and f(s ⊕ m) ≤ BestCost then
         begin
            if f(s ⊕ m) < BestCost then
            begin
               BestCost ← f(s ⊕ m)
               M' ← ∅
            end
            M' ← M' ∪ m
         end
      end
   end
   return M'
end
```

Fig. 2.13. Tabu search version of GenerateLocalMoves

2.3.3 Memory Strategies

2.3.3.1 Tabu Search

The strategy of a tabu search is to keep a list of previously visited solutions to ensure the search does not visit the same solution twice (i.e. the solutions on the list become *tabu* or forbidden). When a local minimum is encountered, the search will escape by selecting the best *alternative* solution to the minimum [Glover, 1989]. However, it is usually impractical to store all visited solutions, as the list can become large and diffi-

cult to search. Instead a list is usually kept of the most recent *moves* made in the search. By forbidding the undoing of an existing move, a search can still avoid revisiting the same solution. Generally lists of forbidden moves have a fixed length, meaning that after a certain number of iterations a move is dropped from the list and becomes allowable again (otherwise all possible moves can become tabu, and the search will become stuck). However, a fixed-length list can lead to the possibility of cycling (i.e. the same *series* of moves are repeated). Therefore the choice of list length is important - long enough to avoid cycles, but short enough to avoid running out of possible moves. Empirical studies have shown the optimum list length to differ between problems but to remain fairly stable for the same problem type and size [Glover, 1989]. An example CSP tabu search algorithm is shown in figure 2.13. Here the tabu list is implemented using the LastUse(m) function which returns the iteration in which move m was last used, hence if *TotalMoves* - LastUse(m) > *MaxListLength* then m is tabu (MakeLocalMove for tabu search follows figure 2.5, i.e. a move is randomly selected from M'):

Within the literature, a number of more complex tabu search techniques have been developed, a selection of which are introduced in the following subsections:

Aspiration Level Conditions: The inclusion of a move on a tabu list can mean that many possible solutions become tabu, not just the solution created by the move. Later in the search, the same move may be considered and rejected, even though (because other variable values have changed) it could lead to a new and better solution. The inclusion of an aspiration level is designed to remedy this situation and allow the repetition of a tabu move if it results in a better than previously possible solution. This can be implemented by comparing the cost of the solution produced by the tabu move with the least cost solution found so far in the search - if the move results in a lower cost then it is accepted [Hertz *et al.*, 1995]. More complex aspiration level schemes have been developed (e.g. see [Glover, 1989]), but the basic principle remains the same: tabu moves can be accepted so long as they *aspire* to produce solutions that improve upon a defined cost threshold.

Reactive Search: Battiti extended the idea of a fixed length tabu list, by proposing the length of the tabu list should vary according to the current state of the search [Bat-

titi, 1995]. This reflects the idea that a search should concentrate in promising areas, but also be able to *diversify* once an area no longer appears promising. Hertz *et al*. [1995] proposed adjusting the cost function so that solutions with similar characteristics are either penalised or rewarded depending on whether concentration or diversification is desired. Both schemes represent a *reactive* search, that changes behaviour through *feedback* about the current state of the search. Battiti's Reactive Tabu Search (RTS) operates on the principle that the more a search attempts to re-visit the same solution, the more diversification is required (repeatedly visiting the same solutions indicates the search has found a local minima and is having trouble escaping). Conversely, the fewer repetitions there are, the more concentrated a search has to be in order not to miss a promising alternative. In Battiti's method, diversification is controlled by allowing the length of the tabu list to grow as more repetitions are encountered, and to shrink as the number of repetitions decrease.

Cancellation Sequences: Glover [1990] extended the idea of keeping a list of tabu moves, with the idea of cancellation sequences. The insight behind this is that a solution is not necessarily revisited unless a move is made, and then reversed, without any intervening moves. If there are intervening moves, then the whole *sequence* of moves also have to be undone for the same solution to re-occur. A cancellation sequence (C-Sequence) is recognised when a move is made that undoes a previous move. Instead of making the move tabu, the previous move is *cancelled*, and all moves between the cancelled move and the current move are added to a C-Sequence. Only if a C-Sequence is empty does the current move become tabu.

2.3.3.2 HSAT, NOVELTY and RNOVELTY

HSAT was an early variation of GSAT proposed by Gent and Walsh [1993] that exploited the idea of memory for tie breaking. Here, if there is a choice of least cost domain values for a move, the value that was used longest ago is chosen. More recent versions of WSAT [McAllester *et al*., 1997] also have incorporated the HSAT idea to avoid revisiting previous solutions by keeping track of when a variable was last ‘flipped’. This idea is analogous to a tabu search but uses a simpler mechanism: when choosing a move (flip) to fix a constraint (clause), the least cost move is selected *unless* this move uses the most recently instantiated domain value (in comparison to the

the set of domain values available to fix the constraint). In this case the second best cost move may be accepted (depending on the heuristic). NOVELTY accepts the second best move with probability p ($0 \leq p \leq 1$), whereas RNOVELTY additionally allows n (the difference in cost between the best and second best moves) to influence the selection (see figures 2.14 and 2.15). By considering n, RNOVELTY applies another idea from simulated annealing, where the probability of acceptance is also conditioned by the size of cost increase caused by a move.

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
   if TotalMoves modulus RandomMovePeriod = 0 then s ← s ⊕ random move
   BestCost ← ∞, SecCost ← ∞
   randomly select a violated constraint c
   for each v_next ∈ c do
   begin
      d_curr ← current domain value of v_next
      for each d ∈ D_next | d ≠ d_curr do
      begin
         m ← {v_next, d}
         if (f(s ⊕ m) = BestCost and LastUse(m) < LastUse(m_best)) or
            f(s ⊕ m) < BestCost then
         begin
            SecCost ← BestCost
            m_best ← m
            BestCost ← f(s ⊕ m)
         end
         else if (f(s ⊕ m) = SecCost and LastUse(m) < LastUse(m_sec)) or
                 f(s ⊕ m) < SecCost then
         begin
            m_sec ← m
            SecCost ← f(s ⊕ m)
         end
      end
   end
   if m_best does not undo most recent change of all v_next ∈ c then m_sec ← ∅
   return m_best ∪ m_sec
end
```

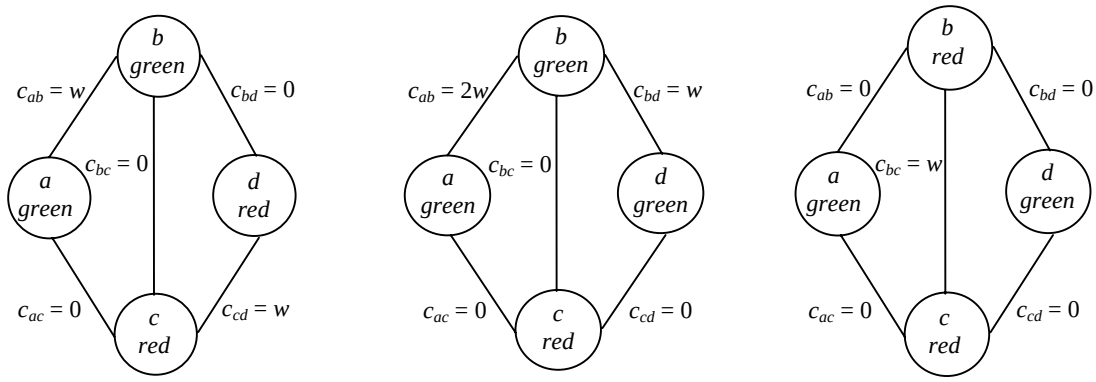
Fig, 2.14. RNOVELTY version of GenerateLocalMoves

procedure MakeLocalMove(*s, M′ , TotalMoves*)
begin
  select best move m_{best} from M'
  select second best move m_{sec} from M'
  if $m_{sec} \neq \varnothing$ **then**
  begin
    $n = f(s \oplus m_{sec}) - f(s \oplus m_{best})$
    r = random number between 0 and 1
    if ($n \leq MinDiff$ **and** $r < 2p$)
    or ($n > MinDiff$ **and** $r < 2p - 1$) **then** $m_{best} \leftarrow m_{sec}$
  end
  $s \leftarrow s \oplus m_{best}$, LastUse($m_{best}$) $\leftarrow$ *TotalMoves*, *TotalMoves* $\leftarrow$ *TotalMoves* + 1
end

Fig. 2.15. RNOVELTY version of MakeLocalMove

2.3.4 Weighting Strategies

Constraint weighting schemes solve the problem of local minima by adding weights to the cost of violated constraints. These weights permanently increase the cost of violating a constraint and so change the shape of the cost surface until the minimum can be exceeded [Morris, 1993]. This is illustrated in the following example of an over constrained graph colouring problem:



(a) Local minimum, cost 2w *(b) After Weighting, cost 3w* *(c) Final solution, cost w*

Fig. 2.16. Using constraint weighting for graph colouring

In the graphs of figure 2.16 the nodes *a, b, c* and *d* represent the variables or areas to be coloured, each having two domain values {red, green}, and the arcs c_{ab}, c_{ac}, c_{bc}, c_{bd} and c_{cd} represent the constraints $a \neq b$, $a \neq c$, $b \neq c$, $b \neq d$ and $c \neq d$ respectively. Given that each constraint violation adds a cost of w to the solution, the situation in figure

2.15a represents a local minimum of cost 2*w*. A constraint weighting algorithm could then add a further weight *w* to each violated constraint increasing the cost of the solution to 4*w*. This alters the problem so that a choice of lower cost moves become available. Figure 2.15b shows the effect of changing the value of *d* to green, causing c_{bd} to be violated at a cost increase of *w*, but satisfying c_{cd} at a cost *decrease* of 2*w*. From figure 2.15b the best cost decreasing move is to change *b* to red, leading to the (optimal) solution in figure 2.15c where only one constraint, c_{bc}, is violated:

Figure 2.17 shows a CSP constraint weighting algorithm for solving CSPs *and* PCSPs. Because constraint weighting makes moves in a weighted cost space, but is looking for a solution in an unweighted cost space, two cost functions are required:

- $f_w(s)$ finds the weighted cost of *s* and is used for move selection.
- $f(s)$ finds the original (unweighted) cost of *s* and is used to test whether the desired cost level has been reached in the local search framework of figure 2.3.

Otherwise the algorithm follows the basic hill climbing strategy of figures 2.4 and 2.5, with the addition of adding weights at each local minimum ($M' = \varnothing$).

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
   M' ← ∅, BestCost ← fw(s) - δ
   for each v_i ∈ V do if v_i in constraint violation then
   begin
      d_curr ← current domain value of v_i
      for each d ∈ D_i | d ≠ d_curr do
      begin
         m ← {v_i, d}
         if fw(s ⊕ m) ≤ BestCost then
         begin
            if fw(s ⊕ m) < BestCost then
            begin
               BestCost ← fw(s ⊕ m)
               M' ← ∅
            end
            M' ← M' ∪ m
         end
      end
   end
   if M' = ∅ then increase weights on all violated constraints
   return M'
end
```

Fig. 2.17. Constraint weighting version of GenerateLocalMoves

2.3.4.1 Developments in Constraint Weighting

Since the initial application of constraint weighting to solve satisfiability problems, [Frank, 1996; Frank, 1997] suggested several performance enhancing modifications to the original algorithm, including updating weights after each move (instead of at each minimum), using different functions to increase weights and allowing weights to decay over the duration of the search. [Cha and Iwama, 1996] produced significant performance improvements on CNF satisfiability problems with their Adding New Clauses (ANC) heuristic, which instead of adding weights at a local minimum, adds a new clause for each violated clause (the new clause being the resolvent of the violated clause and one of it's neighbours). [Castell and Cayrol, 1997] suggest an extended weighting algorithm called Mirror which, in addition to weighting, has a scheme for 'flipping' variable values at each local minimum. However, both ANC and Mirror are domain dependent techniques, ANC relying on constraints being represented as clauses of disjunct literals and Mirror requiring Boolean variables (Mirror also only appears useful for a small class of satisfiability problems). More recently, Wah [Wah and Shang, 1997; Wu and Wah, 1999] has done significant work in providing a mathematical framework for constraint weighting based on the idea of discrete Lagrangian multipliers, called Discrete Lagrangian Methods (DLM). Wah's work has also introduced several variations on the basic weighting scheme, including the rescaling of weights during the search [Wah and Shang, 1997], the introduction of tabu lists and a special weighting scheme that places extra weight on frequently violated constraints [Wu and Wah, 1999].

In the broader domain of AI, constraint weighting heuristics have been applied to neural networks [Davenport *et al.* 1994], genetic algorithms [Bowen and Dozier, 1996], timetabling problems [Cha *et al.*, 1997; Thornton and Sattar, 1999] and staff scheduling [Thornton and Sattar, 1997]. Specialised constraint weighting algorithms have also been proposed for over-constrained problems with hard (mandatory) and soft (desirable) constraints (e.g. [Cha *et al.*, 1997; Voudouris and Tsang, 1996; Thornton and Sattar, 1998b]). In particular, Guided Local Search (GLS) [Voudouris and Tsang, 1996] broadens the idea of weighting constraints to the idea of penalising 'features' in the problem and introduces a utility function to guide the weighting of individual features (GLS is considered further in Chapters 4, 5 and 6).

2.3.4.2 Constraint Weighting and Tabu Search

Alternative constraint weighting schemes have been independently developed to enhance the performance of tabu search strategies in over-constrained environments [Gendreau *et al.*, 1994]. These schemes apply weights to different groups of constraints, increasing the weight if a group is consistently in violation and decreasing the weight if the group is consistently satisfied. Generally the weights cycle between upper and lower bounds and assist the search to maintain the differing importance of constraints while encouraging the search to diversify (i.e. not to get fixed on solutions that satisfy the more important constraints). The issue of solving over-constrained problems with differing constraint priorities is explored further in Chapter 6.

Minima Avoiding Strategy	Strategy Method	Example Techniques
None	Accept the best local move that does not increase the solution cost (hill-climbing)	Min-Conflicts Heuristic [Minton *et al.*, 1992]; Simplex Method [Dantzig, 1963]
Restart	Terminate unsuccessful search at stopping condition and restart from a new initial solution	GSAT [Selman *et al.*, 1992], Averaging-In [Selman and Kautz, 1993], EFLOP [Yugami *et al.*, 1994]
Stochastic	Accept non-improving moves according to a given probability distribution	Simulated Annealing [Abramson, 1992], WSAT [Selman *et al.*, 1994]
Memory	Use recorded characteristics of previously visited solutions to avoid revisiting these solutions	Tabu Search [Glover, 1989], HSAT [Gent and Walsh, 1993] NOVELTY, RNOVELTY [McAllester *et al.*, 1997]
Weighting	Place weights on unsatisfied constraints to bias the search to satisfy these constraints	Breakout [Morris, 1993], Clause Weighting [Selman and Kautz, 1993], DLM [Wah and Shang, 1997], GLS [Voudouris and Tsang, 1996]

Table 2.1. A local search taxonomy

2.4 Summary

The chapter firstly introduced the basic principles of constraint satisfaction and constructive search. Then a taxonomy of local search techniques was developed, based on the methods used to escape or avoid local minima. Using a common local search framework and an explanation of the neighbourhood searching heuristic, four fundamental local search strategies were introduced and then specified by redefining the functions called from the general local search algorithm of figure 2.3. The resulting local search taxonomy is summarised in table 2.1. This taxonomy shows that a large range of local search techniques can be simply explained as the application and combination of four basic ideas: restart when the search looks unpromising, add some randomness into the selection of moves, avoid visiting previous solutions and add weights to constraints that are repeatedly violated.

Chapter 3

Modelling Realistic Problems

In this chapter we examine the issues involved in transforming complex realistic problems into a local search CSP framework. To do this we look at two example real world scheduling problems: university timetabling and nurse rostering (problems that will later be used in the empirical studies). We are specifically interested in developing a general local search approach that can efficiently solve complex problems without the need of domain dependent heuristics and move operators. This leads us to propose an array-based domain representation and array-based resource constraints that internally represent and count domain value usage.

3.1 Specific and General Solutions

It seems to be an axiom of computer science that a general purpose algorithm will solve more problems less efficiently than a problem specific algorithm that employs the best heuristics and data structures available for a given situation [Minton, 1996]. However, the time and effort involved in developing problem specific solutions means a general approach is often more practical. In particular, declarative techniques such as Constraint Logic Programming (CLP) offer the promise of simply describing a problem and obtaining an answer without specifying an algorithm, heuristic or data structure.

A major issue in constraint satisfaction is exactly how a problem is represented. By defining the domain of a queen in the *n*-queens problem as the whole board we obtain a much harder problem than defining the domain as a row (see Chapter 1). As problems become more complex and realistic the number of possible representations also

grows, and the issue of efficiency becomes more important. While a large range of problems can be modelled as discrete domain CSPs and solved using simple binary or non-binary arithmetic and tree constraints, in many circumstances this can result in complex models that are time consuming to solve. This has been recognised by the CLP community and has resulted in the development of more efficient specialised constraints such as `alldifferent` and `cumulative` [Marriott and Stuckey, 1998]. By developing domains and constraints that exploit specific situations, a general CSP approach can also be made more efficient. This chapter explores several extensions to the standard CSP formulation that proved useful in solving two specific scheduling problems. While retaining the basic structure of a CSP algorithm (i.e. variables instantiated with domain values and checked with constraints), we examine binary vs. non-binary constraint representations and show how array-based domain values can make problems easier to solve. In addition we examine complex constraints and domain values that can encode more efficient move operators. This leads us to propose specialised local search `alldifferent`, `block` and `gap` constraints. Firstly, we introduce the problems that are the basis of our further discussion:

3.2 Problem Descriptions

The two problems considered in this chapter are based on actual organisations: the timetabling problem models the situation at the Gold Coast Campus of Griffith University and the nurse rostering problem is taken from two wards at the Gold Coast Hospital, Southport, Queensland. The constraints defined below either match or exceed the current standards of the two organisations and are designed to produce realistic working solutions rather than approximate or idealised answers:

University Timetabling: A university timetabling problem consists of a set T of teaching staff, $\{t_1, t_2, ..., t_{tmax}\}$, a set R of rooms, $\{r_1, r_2, ..., r_{rmax}\}$, a set G of student groups, $\{g_1, g_2, ..., g_{gmax}\}$ and a set C of classes, $\{c_1, c_2, ..., c_{cmax}\}$. The objective of the problem is to assign to each class c_k a staff member t_i, a room r_j, a subset of student groups $g_C \subseteq G$ and a time interval $start_k$ .. end_k such that the following constraints are satisfied ($\forall c_k \in C$):

1. staff member t_i is qualified to teach class c_k
2. staff member t_i is available to teach during interval $start_k$ .. end_k
3. all groups in g_C are enrolled in class c_k
4. room r_j can hold class c_k
5. the duration of class $c_k = end_k - start_k$
6. non-elective enrollments for all groups $g_l \in G$ are satisfied
7. staff member t_i does not teach more than $maxweek_{ti}$ hours per week
8. no $t_i \in T$, $r_j \in R$ or $g_l \in G$ is assigned two classes with overlapping time intervals
9. no staff member $t_i \in T$ teaches more than $maxblock_{ti}$ hours of consecutive classes
10. no group $g_l \in G$ attends more than $maxblock_{gl}$ hours of consecutive classes

In addition the following 'soft' (desirable but not mandatory) constraints are defined:

1. interval $start_k$ .. end_k is during a preferred teaching interval for staff member t_i
2. same day gaps between classes for all groups $g_l \in G$ do not exceed $maxgap_{gl}$
3. same day gaps between classes for all staff $t_i \in T$ do not exceed $maxgap_{ti}$
4. elective enrollments for all groups $g_l \in G$ are satisfied
5. lectures precede tutorials and laboratories in the same subject

Nurse Rostering: A nurse rostering problem consists of a set N of nursing staff, $\{n_1, n_2, ..., n_{nmax}\}$ and a set W of work shifts, $\{w_1, w_2, ..., w_{wmax}\}$. The objective of the problem is to assign to each shift w_i a nurse n_j and a time interval $start_k$ .. end_k such that the following constraints are satisfied:

1. for each time interval k, for each nurse skill level m, the number nurses num_{mk} with skill level m, assigned to interval k, is bounded by:
   $MinStaffNeeded_{mk} \leq num_{mk} \leq MaxStaffAllowed_{mk}$
2. $\forall w_i \in W$, if shift w_i is assigned nurse n_j in interval k, n_j must be available for interval k
3. $\forall n_j \in N$, n_j must work exactly $TotalIntervals_{nj}$ time intervals per roster
4. no nurse $n_j \in N$ works more than $TotalNights_{nj}$ night time intervals per roster
5. no nurse $n_j \in N$ works more than one time interval per day
6. no nurse $n_j \in N$ works more than $MaxOnBlock_{nj}$ days without a day off
7. no nurse $n_j \in N$ works less than $MinOnBlock_{nj}$ days without a day off

8. no nurse $n_j \in N$ works more than $NightBlocks_{nj}$ consecutive night time intervals
9. no nurse $n_j \in N$ has a block of less than $MinOffBlock_{nj}$ consecutive days off
10. $\forall n_j \in N$, n_j must have at least $NightGap_{nj}$ hours break after a night shift
11. $\forall n_j \in N$, n_j must have at least $DayGap_{nj}$ hours break after a day shift

Again the following 'soft' constraints are defined:

1. days off should be preceded by a shift with interval k such that $end_k \leq end_{desired}$
2. days off should be followed by a shift with interval k such that $start_k \geq start_{desired}$
3. no nurse $n_j \in N$ works more than $DesiredMaxBlock_{nj}$ days without a day off
4. no nurse $n_j \in N$ works less than $DesiredMinBlock_{nj}$ days without a day off
5. $\forall w_i \in W$, if w_i is assigned nurse n_j and interval k, k should not overlap a requested time off interval for nurse n_j

Given the problem definitions, we now look at issues arising from modelling these problems as CSPs. Firstly we consider the transformation of non-binary constraints into equivalent binary representations:

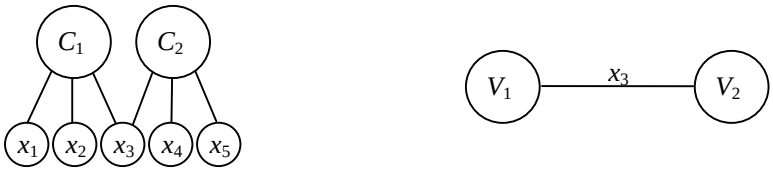
3.3 Binary vs. Non-Binary Representation

Much of the work in constraint satisfaction has concentrated on binary CSPs, i.e. problems where constraints only involve two variables. Binary constraints can be simply expressed and processed, allowing for concentration on the underlying problem rather than the details of representing specific constraints. Although it is well known that any non-binary problem can be transformed into an equivalent binary representation [Rossi et al., 1990], the question as to whether such transformations are efficient or desirable has only recently been addressed [Bacchus and van Beek, 1998].

3.3.1 Transforming Non-Binary CSPs

From the original definition of a CSP (Section 2.1) we know that a constraint can be represented as a relation where each tuple is a combination of variable values that satisfy the constraint. In the *dual graph* method for transforming a non-binary CSP into an equivalent binary representation [Rossi et al., 1990], each non-binary constraint

becomes a variable whose domain is the original constraint relation. Binary equality constraints then exist between all transformed variables that share variables from the original problem. For example, consider two constraints C_1 and C_2, such that C_1 constrains variables x_1, x_2 and x_3 to be equal and C_2 constrains variables x_3, x_4 and x_5 to be not equal, and where x_1 ... x_5 share the same domain {0,1,2}. In this case the non-binary problem can be illustrated in figure 3.1a.



(a) (b)

Fig. 3.1. Non-binary and binary constraint graphs

The transformed problem, shown in Figure 3.1b, contains two variables V_1 and V_2 connected by a single binary constraint arc labelled with x_3. V_1 is the result of transforming C_1 and has the domain of all the {x_1, x_2, x_3} tuples that satisfy C_1 (i.e. {0,0,0}, {1,1,1}, {2,2,2}). Similarly V_2 is the result of transforming C_2 and has the domain of all the {x_3, x_4, x_5} tuples that satisfy C_2 (i.e. {0,1,2}, {0,2,1}, {1,0,2}, {1,2,0}, {2,0,1}, {2,1,0}). Then as V_1 and V_2 share variable x_3 in the original problem, a binary constraint is added ensuring the domain elements corresponding to x_3 are equal (i.e. if V_1 = {0,0,0} then the only values from C_2 that satisfy the x_3 constraint are {0,1,2} and {0,2,1}).

Other transformation techniques exist, the best known being the *hidden variable* method [Bacchus and van Beek, 1998]. A hidden variable transformation preserves the original variables and their domains, but creates additional hidden variables to represent each non-binary constraint. A hidden variable has domain values which identify tuples in an original non-binary constraint. Hence if H_1 is a hidden variable representing C_1, a value in H_1 of 2 would correspond to tuple 2 in C_1 (i.e. {1,1,1}). H_1 then has three binary constraints with x_1, x_2 and x_3 respectively, ensuring the values of x_1, x_2 and x_3 conform to the tuple indicated in H_1 (in this case if H_1 = 2 then H_1 is satisfied only when x_1 = 1, x_2 = 1 and x_3 = 1). Alternatively, the hidden variable domain can directly represent the satisfying tuples of the original constraint, in the same way as a dual variable [Stergiou and Walsh, 1999]. Here the only difference be-

tween techniques is in the type of binary constraint used (either between dual variables, or between hidden variables and the corresponding original variables). The consequences of non-binary transformations for large and realistic constraints are examined in the next section:

3.3.2 Domain Size Issues in Non-Binary Transformations

As a practical example of non-binary to binary transformation we consider the first constraint in the rostering problem defining the number and level of staff required for each shift (an instance of this constraint is that at least 7 and at most 9 registered nurses are needed on the first Monday shift between 7.30 am and 3.30 pm). Let us assume each possible time interval for each nurse is a variable with a {0,1} domain, where 1 means the nurse works the interval and 0 means the nurse does not work the interval. In this case, our example constraint would have all Monday 7.30 am to 3.30 pm time intervals of available registered nurses as variables. The constraint would then be satisfied if the sum of all the variables is greater than 6 and less than 10. If we assume there are 20 registered nurses available during the interval, the constraint can be represented arithmetically as:

$$6 < x_{1,1} + x_{1,2} + \ldots + x_{1,20} < 10$$

where x_{ij} is a {0,1} variable and where i identifies a time interval and j identifies a nurse. The non-binary constraint graph for this constraint is also pictured in Figure 3.2:

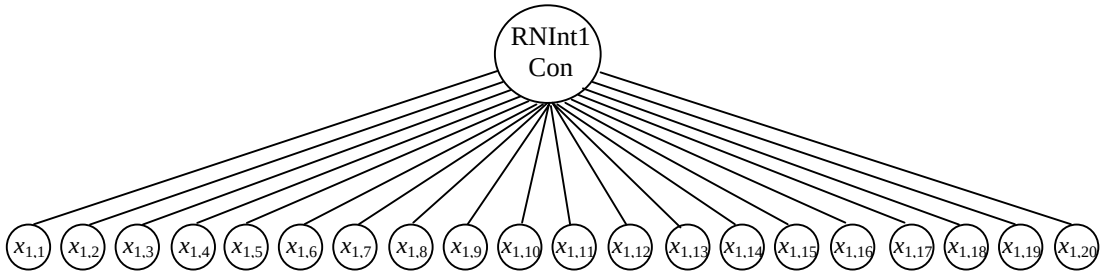


Fig. 3.2. A non-binary staff requirement constraint

If we perform a dual graph transformation on this non-binary constraint, we obtain a variable v_1 whose domain is all the possible ways between 7 to 9 of the 20 available nurses can be assigned to interval 1. Hence, an example domain value with nurses 4, 6, 9, 13, 15, 16 and 17 working interval 1 would be:

$$0,0,0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,0,0,0$$

and the total number of domain values for v_1 is given by:

$$\frac{20!}{7!13!} + \frac{20!}{8!12!} + \frac{20!}{9!11!} = 371,450$$

Given such a large domain, the space requirements for representing the problem become significant. A standard binary CSP algorithm models a binary constraint between two variables *x* and *y* by creating a two-dimensional array such that each element e_{ij} in the array is 1 if domain value *i* for *x* and domain value *j* for *y* satisfy the constraint (otherwise $e_{ij} = 0$). Such *extensional* constraint representations are only practical for small domain problems. A nurse roster can have over 200 non-binary staff requirement constraints, translating into 200 variables each with several thousand (or hundred thousand) domain values. These variables in turn are involved in binary constraints with at least 35 other variables, each one of which can again have several thousand domain values. Conservative estimates of the space required to represent the problem with 1 byte array elements soon reach 1,000 Gb.

The alternative is to represent the problem *intensionally*, i.e. by using run-time functions to construct valid tuples for the original non-binary constraints and calculating whether these tuples satisfy the binary constraints in the transformed problem (for example, ILOG® Solver uses hidden variable encoding to represent binary constraints). Clearly, the construction of a valid tuple involves evaluating the original non-binary constraint and so is the *same as solving the original non-binary problem*. In fact the transformation of a non-binary problem can be considered as an off-line application of non-binary constraints in order to generate satisfying tuples and so avoid the need of applying non-binary constraints during the solution process. From this it follows that transformations of non-binary constraints that result in very large domain sizes are unlikely to be useful.

3.3.3 Partial Non-Binary to Binary Transformation

The issue of problem representation has concentrated on transforming non-binary into binary constraints, so that binary CSP techniques can be used [Rossi et al., 1990]. As Bacchus and van Beek [1998] show, certain problems can be solved more efficiently using binary constraints while others are better represented using non-binary con-

straints. Our analysis in the previous section shows the transformation of non-binary constraints can be impractical due to the size of the resulting domain. However we are not required to represent constraints in a system as either all binary or all non-binary – a mix of both constraint representations is also possible.

For instance, consider the remaining non-binary constraints in the nurse rostering problem. We have already rejected the transformation of the staffing levels constraint (constraint 1) due to the resulting domain size. Now we consider constraints 2 to 11 from section 3.1: each of these constraints partially defines the allowable combinations of time intervals for each nurse. For example constraint 3 defines the total time intervals each nurse can work in a roster. Using our previous notation, each interval for each nurse is a {0,1} variable, so we can represent constraint 3 as the sum of all time interval variables for each nurse. Assuming nurse 1 must work 8 intervals and the roster is 42 intervals long, then the constraint for nurse 1 would be (where x_{ij} is a variable and where i identifies a time interval and j = identifies a nurse):

$$x_{1,1} + x_{2,1} + \ldots + x_{42,1} = 8$$

with a domain size of:

$$\frac{42!}{8!34!} = 118,030,185$$

Again processing such a large domain would be impractical, but this domain can be significantly reduced if we apply the remaining time interval constraints. For instance, applying constraint 5 (no nurse should work more than one time interval per day) reduces the domain size for nurse 1 to 4,546,773 tuples. If we assume nurse 1 cannot work less than 3 or more than 7 consecutive shifts, must have day off periods at least 2 days long and is scheduled to work at most 3 night shifts in a single block then the domain size can be reduced to 32,960 tuples. If we simplify the problem by only allowing two time intervals per day (day and night shift) the domain can be further reduced to 376 tuples. To clarify this, the table 3.1 shows an example domain value for nurse 1 (the last row represents the actual variable values $x_{1,1}$, $x_{2,1}$, ..., $x_{42,1}$):

Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Day	*Day*	*Day*	*Off*	*Off*	*Off*	*Day*	*Day*	*Night*	*Night*	*Night*	*Off*	*Off*	*Off*
100	100	100	000	000	000	100	100	001	001	001	000	000	000

Table 3.1. An example tupled nurse variable domain

In effect, we have treated all the time interval constraints as one complex non-binary constraint and transformed this constraint into a dual graph binary representation i.e. we have generated a tupled variable whose domain is all possible values that satisfy the original non-binary constraints. This is an example of a *desirable* non-binary to binary transformation, because:

1. The new tupled domains are small enough to be represented extensionally.
2. The dual variable removes all the time interval constraints from the problem, meaning both the number of consistency checks and the size of the search space are reduced (as infeasible time interval combinations are no longer possible).

We now have a CSP that has been *partially* transformed from a non-binary to a binary representation, with a set of non-binary staffing level constraints that constrain the original `{0,1}` variables (from section 3.2.2), and a set of tupled variables representing the allowable permutations of time intervals for each nurse. The question now is how to combine these representations into a single problem.

3.3.4 Defining Constraints for Tupled Domains

The combination of binary and non-binary representations for the rostering problem can be achieved if we define non-binary constraints between the original `{0,1}` variables that are now represented *within* the tuples of the time interval variables. For example, consider a rostering problem consisting of 3 nurses and 4 time intervals with each nurse working 3 time intervals in the roster. Figure 3.3 shows each of the 4 possible time interval combinations for a nurse in such a problem (e.g. domain value `1` represents a nurse working time intervals 2, 3 and 4):

time interval

domain value	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

Fig. 3.3. Nurse domain values for simplified problem

If we consider each nurse as a variable v_i ($i = 1..3$) with a tupled domain of time intervals, then a problem solution will contain a domain value for each nurse:

		time interval			
		1	2	3	4
nurse variable	v_1	1	0	1	1
	v_2	1	0	1	1
	v_3	1	1	1	0
total nurses on duty		3	1	3	2

Fig. 3.4. Example solution for simplified problem

Now consider the staffing constraint that at least 2 nurses are required on duty in each time interval (in figure 3.4, this constraint is violated in time interval 2). A standard mathematical representation of this problem (e.g. [Warner, 1976]) would use $\mathbf{a}_{ij}$ vectors to represent each tupled domain value, where i is the domain value index and j is the variable (nurse) index (i.e. $\mathbf{a}_{11} = (0,1,1,1)$) and a $\mathbf{b}$ vector to represent the minimum staffing requirements for each time interval (i.e. $\mathbf{b} = (2,2,2,2)$). The problem is then solved using $\{0,1\}$ decision variables, X_{ij}, such that $X_{ij} = 1$ if variable j gets domain value i (otherwise $X_{ij} = 0$), as shown in the following constraint definitions:

$$\sum_{j=1}^{j=3} X_{ij} = 1, \quad i = 1..4$$

$$\sum_{i=1}^{i=4} \sum_{j=1}^{j=3} \mathbf{a}_{ij} X_{ij} \geq \mathbf{b}$$

These constraints can then be transformed into a set of linear inequalities that can be solved using standard arithmetic constraints. In this case the X_{ij} constraints for each nurse would be (indicating a nurse can only work one schedule):

$$X_{11} + X_{21} + X_{31} + X_{41} = 1$$
$$X_{12} + X_{22} + X_{32} + X_{42} = 1$$
$$X_{13} + X_{23} + X_{33} + X_{43} = 1$$

and the $\mathbf{a}_{ij}X_{ij}$ constraints for each time interval would be (indicating there must be at least two nurses on duty in each interval):

$$X_{21} + X_{31} + X_{41} + X_{22} + X_{32} + X_{42} + X_{23} + X_{33} + X_{43} \geq 2$$
$$X_{11} + X_{31} + X_{41} + X_{12} + X_{32} + X_{42} + X_{13} + X_{33} + X_{43} \geq 2$$
$$X_{11} + X_{21} + X_{41} + X_{12} + X_{22} + X_{42} + X_{13} + X_{23} + X_{43} \geq 2$$
$$X_{11} + X_{21} + X_{31} + X_{12} + X_{22} + X_{32} + X_{13} + X_{23} + X_{33} \geq 2$$

In effect, we have transformed the tupled problem back to an atomic valued problem and added back constraints to ensure each nurse only works one schedule ($\sum X_{ij} = 1$). If we solve this problem using a standard CSP algorithm (i.e. by instantiating the X_{ij} variables and testing the constraints) we will incur the extra cost of evaluating the $\sum X_{ij} = 1$ constraints. This cost is not incurred for a tupled domain because a variable (nurse) can only have one domain value (schedule) at a time. Therefore a tupled representation should perform fewer consistency checks as it will avoid trying multiple schedule allocations. However, to evaluate the non-binary staffing constraints using tupled variables, we need to specify the atomic value in the tuple to which a constraint applies. A general way to approach this problem is to treat the tupled domain as an *array* of ordered atomic values from which a single element can be referenced using an index. In this way, the time interval constraints for the tupled representation can be expressed in the following inequalities:

$$v_1[1] + v_2[1] + v_3[1] \geq 2$$

$$v_1[2] + v_2[2] + v_3[2] \geq 2$$

$$v_1[3] + v_2[3] + v_3[3] \geq 2$$

If we compare this system of constraints to the previous atomic valued equivalent we can see the problem is captured more simply and so can be solved more efficiently. The cost of the tupled representation is that a constraint check must look up an array value rather than reading a domain value directly. While this is not a significant cost, it does require the development of constraints that can operate on array domains.

3.3.5 Lessons for General Problems

Although the preceding discussion has concentrated on a particular problem, the underlying concepts can be applied more generally. The process of transforming a non-binary to a binary constraint (using the dual graph method) is equivalent to *solving* the non-binary CSP represented by the original constraint. The domain of the new dual variable then becomes the set of all solution tuples for the original non-binary CSP. Many complex problems can be more efficiently solved by recognising and solving smaller sub-problems and then using these answers to construct a complete answer to the original problem [Freuder and Hubbe, 1995]. By definition, a CSP solution is a tuple of domain values (one for each variable), and consequently a complete enumera-

tion of solutions to a sub-CSP can be represented in the main problem as a dual variable with an array-based domain and solved using array processing constraints.

Our previous discussion has shown a tupled domain can be replaced with atomic values using vectors and decision variables. However this *always* requires additional constraints to ensure only one value from each original tuple is instantiated. An array-based representation avoids the cost of these constraints and is therefore likely (depending on the problem structure) to produce a smaller representation that can be solved more efficiently.

3.4 Representing Complex Move Operators

A ‘move’ for a CSP algorithm is achieved either by instantiating an uninstantiated variable with a valid domain value or by replacing the existing domain value with another value from the same domain (Chapter 2). Such moves are simple to implement and keep the CSP approach general. However, for more complex problems, more sophisticated move operators can significantly improve the efficiency of a search (for example the 2-OPT and 3-OPT moves in the Travelling Salesman Problem [Glover, 1989]). To implement a complex move within a general CSP framework we can either define a move operator function and embed it within the CSP algorithm, or we can define the problem variables and domains in such a way that changing a domain value effects the desired move. In looking at the timetabling problem we consider the second approach, firstly because an efficient move can be easily defined and secondly because this maintains the generality of our implementation.

3.4.1 Making a Move in a Timetabling Problem

When solving or fixing a timetabling problem, a human timetabler usually thinks in terms of moving a class from one room/time allocation to another. If rooms, staff and student groups are all represented as variables, then moving a class involves defining a fairly complex move operator that simultaneously changes the domain values of several variables. Alternative moves that change the values of single variables would be less efficient, as they would explore many more infeasible solutions and several such moves would be required to replace the one more complex move. Given we want

to avoid the use of problem specific move operators, the alternative is to look at different ways of modelling the problem:

Decision Variables. One possibility is to again think in terms of zero-one X_{ijk} decision variables. In this case, X_{ijk} = `1` could represent class *i* occurring in room *j* at time *k* and X_{ijk} = `0` would represent class *i not* occurring in room *j* at time *k*. Constraints to avoid clashes for student groups can then be defined using a set of $n \times m$ vectors (where n = number of groups and m = number of time slots) each with p_{nm} elements (made up of all the X_{ijk} variables for group n at time m) such that the sum of elements for each vector equals one. If we substitute classes for time slots, the model can then be used to constrain each group to only attend a particular class once (staff member teaching constraints can be defined in the same way). Using decision variables allows a complex move (in the original problem) to be effected with a single change of domain. However, several disadvantages still remain:

1. Changing the domain value of a decision variable does not move a class from one room/time slot to another, instead it turns a room/time slot for a class on or off. Moving a class would involve turning the current class room/time allocation off and turning another class room/time allocation on i.e. we still require a special move operator.

2. If we leave the problem in it's current form, then additional constraints are required to ensure a class is only scheduled once and that staff members and student groups only attend a class once - this situation would not arise if we were able to represent a class as a single variable.

3. Finally, the zero-one representation will provide poorer guidance to a local search than a representation that moves a class directly from one room/time slot to another. This is because a direct move method evaluates two zero-one moves simultaneously (i.e. it looks at the cost of turning off one slot *and* turning on another).

3.4.2 Defining Array-based Local Search Constraints

One answer to the above problems is to represent each class as a variable and make the variable domain all available room/time allocations. In this model, changing a domain value has the desired effect of moving a class from one room/time slot to another. However, the more complex domain (compared to the zero-one representation) means we also require more complex constraints. Before describing these constraints we first look at the operation of a constraint within a local search algorithm:

Constraint evaluation in a local search. Local search CSP algorithms operate by evaluating the cost change caused by exchanging domain values in the neighbourhood of the current solution (see Chapter 2). Some methods permanently store the cost of using a domain value against each value (e.g. WSAT) and update the relevant costs each time a move is effected. Other techniques calculate the domain value cost 'on-the-fly', i.e. each time the domain value is tested in a solution. The WSAT approach is efficient for problems where a move has relatively small side-effects on the costs of other domain values, whereas 'on-the-fly' testing has a more general application (i.e. it is better when there are large side-effects and performs adequately when there are small side-effects). In either case we need to find the cost change incurred by exchanging the current domain value of a variable with a new value. This can be calculated by summing the cost change for each constraint in which the variable is involved. The general solver developed in the thesis uses an Object-Oriented approach to represent a CSP. Hence we have classes to represent domains, variables and constraints and a constraint engine that implements various local search algorithms. Within this framework, a variable has a data member that points to the list of constraints in which it is involved and a constraint has a method that calculates the cost of exchanging domain values. Using this structure, figure 3.5 shows the constraint engine method that calculates the cost change incurred by exchanging domain value *Old* with domain value *New* in variable *V*:

```
method getCostChange(V, old, new)
begin
   costChange ← 0;
   C ← V.getConstraintList()
   for each constraint C_j ∈ C do
      costChange ← costChange + C_j.testChange(old, new)
   return costChange
end
```

Fig. 3.5. All purpose getCostChange method

Local search `alldifferent` constraints: In the timetabling problem, if we model each class as a variable with a domain of available room/time slots, then the variable will be involved in one large `alldifferent` constraint [Marriott and Stuckey, 1998] which has all other class variables as members (this constraint enforces that no two classes can occur in the *same room at the same time*). Each class *time* value will also be involved in an `alldifferent` constraint for each staff member and student group assigned to the class (ensuring staff members and students do not attend the two classes at the same time). This means a 2-tuple domain value is required of the form (*x*, *y*) where *x* identifies the room/time and *y* identifies the time of a class. Then, given a set of classes c_1 ... c_k, the constraint that no two classes share the same room/time can be defined as: `alldifferent(`c_1`[1], `c_2`[1], ..., `c_k`[1])`. Putting this in the local search structure requires an `alldifferent` constraint that can efficiently find the cost change of exchanging two domain values. One approach is to count the number of times the old and new domain values are used by all the variables in the constraint, as in figure 3.6:

```
method testChange(old, new)
begin
   oldCount ← 0, newCount ← 0, change ← 0
   V ← getVariableList()
   for each variable V_i ∈ V do
      if V_i[index] = old then oldCount ← oldCount + 1
      else if V_i[index] = new then newCount ← newCount + 1
   if oldCount > 1 then change ← -1
   if newCount > 0 then change ← change + 1
   return change
end
```

Fig. 3.6. testChange method for `alldifferent` constraint

However, testChange in figure 3.6 requires up to `2`n + `1` comparisons to evaluate a move (where n = number of variables in constraint), and is significantly less efficient than processing an equivalent zero-one representation. This is because a zero-one problem uses arithmetic sum constraints which can be evaluated in a single comparison (by storing and updating the current sum for each constraint). For example, the zero-one constraint to test whether switching a class *c on* at time *t* causes a clash for staff member *s* requires the simple evaluation:

if sum(X_{ijk} variables) taught by *s* where (*i* = *c* and *k* = *t*) > 0 **then** costChange ← `1`

else costChange ← 0

To compete with a zero-one model our `alldifferent` constraint must find the cost of moving a class in two comparisons (one for removing the old value and one for adding the new value). This requires a direct look-up of the number of times a particular value is instantiated in the current solution and again is best solved using arrays. For example, consider the earlier problem of moving a class taught by staff member *s* to time *t*. An array-based `alldifferent` constraint for staff member *s* would have an array with elements for each time slot in the timetable, such that each element *i* holds a count of the number of classes taught by *s* in timeslot *i*. Given such an array (called domainArray) and a procedure to update the array each time a move is accepted, the revised code for `alldifferent` is shown in Figure 3.7:

```
method testChange(old, new)
begin
   change ← 0
   if domainArray [old] > 1 then change ← -1
   if domainArray [new] > 0 then change ← change + 1
   return change
end
```

Fig. 3.7. Array version of testChange method for `alldifferent`

Local search `block` and `gap` constraints An array-based representation has additional advantages for expressing the more complex constraints in the timetabling problem, i.e. those defining the maximum consecutive *block* of time slots of classes for staff and students (constraints 9 and 10), and those defining the maximum time slot *gap* between classes (‘soft’ constraints 2 and 3). While the `block` constraints can

be *partly* expressed as a series of arithmetic sum constraints in the zero-one model (see Appendix), the `gap` constraints require a specialised non-linear representation (because we have to distinguish between a gap at the beginning of a day and a gap between classes).

Consider the situation for staff member *s*, who is constrained to work no more than *maxBlock* consecutive time slots. Again we can represent all time slots in the timetable as an array and set each array element *i* equal to the number of classes *s* is teaching in time slot *i* (time slot values are *ordered in time*, meaning, for all *i*, time slot *i* precedes time slot *i* + `1`). Now, if we move a class taught by *s* from time *j* to time *k*, we can test the local effect of decrementing the count in array element *j* and incrementing the count in array element *k*. For instance, consider the situation in figure 3.8, where *maxBlock* = `4`:

Timeslot	1	2	3	4	5	6	7	8
Count	0	1	1	1	1	1	0	0

Fig. 3.8. A violated `block` constraint

Here the count from slot `2` to `6` means a block of `5` consecutive time slots is scheduled and consequently the constraint is in violation. Figure 3.9 shows the result of moving a class from slot `4` to `7`:

Timeslot	1	2	3	4	5	6	7	8
Count	0	1	1	0	1	1	1	0

Fig. 3.9. A satisfied `block` constraint

The `block` constraint is evaluated by testing for the presence of a block in the neighbourhood of the time slot that is changed. Moving a class from slot `4` causes the slot `4` count to be decremented to zero, and hence reduces the size of an existing block. The size of this block can be found by counting how many consecutive timeslots either side of slot `4` have a count value greater than zero (see getBlockLength in figure 3.11). Similarly, moving a class to slot `7` causes the slot `7` count to be incremented to one, hence creating or extending an existing block, the size of which can also be calculated by examining the adjacent timeslots. The full logic of the `block` constraint is shown in figures 3.10 and 3.11:

```
method testChange(old, new)
begin
   change ← 0, domainArray [old] ← domainArray [old] - 1
   if domainArray [old] = 0 then
   begin
      getBlockLength(forward, back, old)
      if forward + back + 1 > maxBlock then
         change ← change - (forward + back + 1 - maxBlock)
      if forward > maxBlock then change ← change + (forward- maxBlock)
      if back > maxBlock then change ← change + (back - maxBlock)
   end
   if domainArray [new] = 0 then
   begin
      getBlockLength(forward, back, new)
      if forward + back + 1 > maxBlock then
         change ← change + (forward + back + 1 - maxBlock)
      if forward > maxBlock then change ←change - (forward - maxBlock)
      if back > maxBlock then change ← change - (back - maxBlock)
   end
   domainArray [old] ← domainArray [old] + 1
   return change
end
```

Fig. 3.10. Array version of testChange method for `block` constraint

```
method getBlockLength(forward, back, start)
begin
   forward ← 0, back ← 0
   while domainArray [start + forward + 1] > 0 do forward ← forward + 1
   while domainArray [start - back - 1] > 0 do back ← back + 1
end
```

Fig. 3.11. getBlockLength method for `block` constraint

The `block` constraint requires (*old* block length + *new* block length + 8) comparisons and is competitive with the ($2 \times$ (*maxBlock* + 1)) comparisons needed for a zero-one representation (see Appendix).

A `gap` constraint poses a more complex problem, as a gap exists *between* classes, hence gaps at the beginning or end of the day should be ignored. For example, Figure 3.12 shows an array (as with the `block` constraint) representing a one day schedule for a staff member. Here only the empty slots from 4 to 10 represent an inter-class gap:

TSlot	1	2	3	4	5	6	7	8	9	10	11	12
Count	0	0	1	0	0	0	0	0	0	1	0	0

Fig. 3.12. An unsatisfied `gap` constraint

Assuming the maximum gap between classes (*maxGap*) = `5`, the situation in figure 3.12 is in violation. Figure 3.13 shows the result of moving a class for the staff member from time slot `3` to time slot `8` (hence satisfying the constraint):

TSlot	1	2	3	4	5	6	7	8	9	10	11	12
Count	0	0	0	0	0	0	0	1	0	1	0	0

Fig. 3.13. A satisfied `gap` constraint

Given classes can start at 8 am and finish at 9 pm we cannot decide if a new day has started simply by measuring the gap size (i.e. an 11 hour gap may represent a gap between classes or an overnight break). Therefore the `gap` constraint needs to know how to divide time slots into days or periods - this is achieved with the period function used in getGapLength (see figure 3.15). Given the forward and back gap lengths, the constraint operates in a similar way to the `block` constraint, except the special gap length of `-1` indicates a gap either begins or ends a period. The full `gap` constraint logic is shown in figures 3.14 and 3.15:

```
method getGapLength(forward, back, start)
begin
   forward ← 0, back ← 0
   while forward ≥ 0 and domainArray [start + forward + 1] = 0 do
      if period(start + forward + 1) ≠ period(start) then forward ← -1
      else forward ← forward + 1
   while back ≥ 0 and domainArray [start - back - 1] = 0 do
      if period(start - back - 1) ≠ period(start) then back ← -1
      else back ← back + 1
end
```

Fig. 3.14. getGapLength method for `gap` constraint

```
method testChange(old, new)
begin
   change ← 0, domainArray [old] ← domainArray [old] - 1
   if domainArray [old] = 0 then
   begin
      getGapLength(forward, back, old)
      if forward ≥ 0 and back ≥ 0 and forward + back + 1 > maxGap then
         change ← change + (forward + back + 1 - maxGap)
      if forward > maxGap then change ← change - (forward - maxGap)
      if back > maxGap then change ← change - (back - maxGap)
   end
   if domainArray [new] = 0 then
   begin
      getGapLength(forward, back, old)
      if forward ≥ 0 and back ≥ 0 and forward + back + 1 > maxGap then
         change ← change - (forward + back + 1 - maxGap)
      if forward > maxGap then change ← change + (forward - maxGap)
      if back > maxGap then change ← change + (back - maxGap)
   end
   domainArray [old] ← domainArray [old] + 1
   return change
end
```

Fig. 3.15. Array version of testChange method for `gap` constraint

3.5 Summary

The array-based `alldifferent`, `block` and `gap` constraints were developed specifically to solve a timetabling problem. However, the constraint that a resource cannot be used more than once in a particular time period (represented in the `alldifferent` constraint) is common to many other resource allocation and scheduling problems, e.g. the allocation of ships to berth time slots, teams to match time slots and jobs to machine time slots. In addition, the `block` and `gap` constraints are applicable to any personnel scheduling problems were the number of consecutive hours, shifts or days worked or not worked is important.

In summary, the chapter shows that complex real-world problems can be modelled and solved efficiently as CSPs without the need of domain specific move operators or special heuristics. We present an array-based domain representation for the rostering problem, that allows sum constraints to efficiently operate between array elements of different variables. To represent the timetabling problem, we developed a tupled do-

main for each class that implements a simple class swapping move. To efficiently process these variables we further developed array-based constraints that directly store the level of resource usage for each domain value (in this case time and room/time utilisation).

Chapter 4

Constraint Weighting

In this chapter we examine the behaviour and application of constraint weighting on a range of different problems and in comparison to several other local search techniques. Our aim is to characterise the problem domains for which constraint weighting is most applicable and to evaluate three competing constraint weighting strategies. We extend previous results from satisfiability testing by applying satisfiability techniques to the broader domain of constraint satisfaction and test for differences in performance using randomly generated problems and problems based on the realistic situations described in Chapter 3.

4.1 Background and Motivations

The intensive research into satisfiability testing during the 1990s has produced a set of powerful new local search heuristics. As introduced in Chapter 2, starting from GSAT [Selman *et al.*, 1992], the latest WSAT techniques have raised the ceiling on solving hard 3-SAT problems from several hundred to several thousand variables [Selman *et al.*, 1997]. At the same time, a new class of clause or constraint weighting algorithms have been developed [Morris, 1993; Selman and Kautz, 1993]. These algorithms have proved highly competitive with WSAT (at least on smaller and more difficult problems [Cha and Iwama, 1995]), and have stimulated the successful application of related techniques in several other domains [Thornton and Sattar, 1998a; Bowen and Dozier, 1996; Davenport *et al.*, 1994; Voudouris and Tsang, 1996]. However, since the initial development of constraint weighting, WSAT has evolved new and more powerful heuristics (such as

NOVELTY and RNOVELTY [McAllester *et al.*, 1997]). The improved performance of these heuristics (on hard random 3-SAT problems) brings the usefulness of constraint weighting into question. Consequently, this chapter re-examines constraint weighting in the light of the latest WSAT developments. In order to place our results in a broader context, we also report the performance of tabu and stochastic search algorithms for each of our problem domains. In the process we examine the following questions:

- Are there particular problem domains for which constraint weighting is preferred?
- Does constraint weighting do better on more realistic, structured problems?
- Is there one weighting scheme that is superior on all the domains considered?

The main aim of the chapter is to provide practical guidance as to the relevance and applicability of constraint weighting to the broader domain of constraint satisfaction. Research has already looked at applying WSAT to integer optimization problems [Walser *et al.*, 1998], and applying constraint weighting to over-constrained problems [Thornton and Sattar, 1998b; Voudouris and Tsang, 1996]. However, outside the satisfiability domain, there has been little direct comparison between WSAT and other techniques. The research addresses this by applying WSAT, tabu search and constraint weighting to three CSP formulations: university timetabling, nurse rostering (see Chapter 3) and random binary constraint satisfaction. In addition we explore the behavior of constraint weighting on several classes of satisfiability problem.

The next sections introduce the algorithms used in the study, and then the results for each problem domain are presented. From an analysis of these results we draw general conclusions about the applicability and typical behaviour of constraint weighting.

4.2 Constraint Weighting Algorithms

As introduced in Chapter 2, constraint weighting schemes solve the problem of local minima by adding weights to the cost of violated constraints. These weights permanently increase the cost of violating a constraint, changing the shape of the cost surface so that minima can be avoided or exceeded [Morris, 1993].

Several weighting schemes have been proposed. In Morris's [1993] formulation, constraint weights are initialised to one and violated constraint weights are incremented by one each time a local minimum is encountered. Frank [Frank, 1996; Frank, 1997] adjusts weights after each move and experiments with different initial weights and weight increment functions and with allowing weights to decay over time. Further work has applied constraint weighting to over-constrained problems using dynamic weight adjustment [Thornton and Sattar, 1998b] and utility functions [Voudouris and Tsang, 1996].

In this chapter we are interested in *when* and *what* to weight. Therefore we keep to Morris's original incrementing scheme and explore variations of three of the published weighting strategies:

1. MINWGT: Incrementing weights at each local minimum (based on [Morris, 1993]).
2. MOVEWGT: Incrementing weights when no local cost improving move exists (based on [Frank, 1996] although Frank increments after all moves).
3. UTILWGT: Incrementing weights at each local minimum according to a utility function (based on [Voudouris and Tsang, 1996]).

Voudouris and Tsang's [1996] utility function is part of a more sophisticated algorithm (Guided Local Search or GLS) that handles constraints with different absolute violation costs. They penalise features in a local minimum that have the highest utility according to the following function:

$$utility_i(s^*) = I_i(s^*) \times (c_i / (1 + p_i))$$

where s^* is the current solution, i identifies a feature, c_i is the cost of feature i, p_i is the penalty (or weight) currently applied to feature i and $I_i(s^*)$ is a function that returns one if feature i is exhibited in solution s^* (zero otherwise). In the subsequent problems we assume each feature is a constraint with a cost of one. In this

case the utility function will only select for weighting the violated constraint(s) in a local minimum that have the smallest current weight. Our aim is to test the utility function as a weighting strategy in isolation from the GLS algorithm, to see if it is useful in a more general weighting approach.

The three weighting strategies are tested within the weighting algorithm introduced in Chapter 2 (figure 2.17) with the addition of two new weighting points (shown in figure 4.1).

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
   M' ← ∅, BestCost ← fw(s) - δ
   for each v_i ∈ V do if v_i in constraint violation then
   begin
      d_curr ← current domain value of v_i
      CurrentCost ← BestCost
      for each d ∈ D_i | d ≠ d_curr do
      begin
         m ← {v_i, d}
         if fw(s ⊕ m) ≤ BestCost then
         begin
            if fw(s ⊕ m) < BestCost then
            begin
               BestCost ← fw(s ⊕ m)
               M' ← ∅
            end
            M' ← M' ∪ m
         end
      end
      if MOVEWGT and CurrentCost = BestCost then
         increase weights on all violated constraints containing v_i
   end
   if MINWGT and M' = ∅ then
      increase weights on all violated constraints
   if UTILWGT and M' = ∅ then
      increase weights on all violated constraints with the smallest weight
   return M'
end
```

Fig. 4.1. Three strategies for constraint weighting

4.3 WSAT and Tabu Search Algorithms

The other algorithms considered in this chapter are based on the WSAT and tabu search techniques introduced in Chapter 2. As previously discussed, WSAT avoids local minima by allowing cost *increasing* moves. The algorithm proceeds by selecting violated constraints and then choosing a move which will improve or satisfy the constraint, even when this results in an overall cost increase. The various WSAT schemes differ according to the move selection heuristic employed, here we consider three of the most recently developed variants:

1. BEST: BEST is a stochastic technique which, with probability *p*, will randomly select a move that improves a violated constraint, otherwise it will pick the least cost move that improves the constraint (see figure 2.12).

2. RNOVELTY: RNOVELTY [McAllester *et al.*, 1997] considers both the overall cost of a move and when the move was *last* used. If the best cost move is also the most recently used move then according to a probability threshold and the cost difference between the best and second best moves, the second best cost move may be accepted (see figures 2.14 and 2.15).

3. NOVELTY: NOVELTY is a simplified version of RNOVELTY that does not consider the cost difference between the best and second best available move. Instead we consider choosing the second best move (according to a fixed probability *p*) whenever the best move is also the most recent of the available moves for a constraint.

Finally we implement a *constraint-based* tabu search (TABU) that keeps a list of the *n* most recently changed domain values and ensures that any value on the list is not reused *unless* it leads to a new lowest cost solution (i.e. this is the aspiration criteria explained in Chapter 2). The algorithm also follows the constraint-based neighbourhood selection of WSAT, where GenerateLocalMove (see figure 2.12) randomly selects a violated constraint and then tests all domain values of that constraint.

In the following satisfiability results, the BEST, RNOVELTY, NOVELTY and TABU algorithms directly use the source code developed by Selman, Kautz and McAllester, available at ftp://ftp.research.att.com/dist/ai. These algorithms were then re-coded to solve the other CSP problems with one additional condition: the local neighbourhood is restricted to only include moves that improve the originally selected violated constraint. This condition mirrors the implicit condition in the satisfiability algorithms (i.e. flipping any literal in a violated clause will cause the clause to become true).

4.4 Experimental Results

4.4.1 Satisfiability Results

Research has already demonstrated the superiority of constraint weighting over GSAT and earlier versions of WSAT for smaller randomly generated 3-SAT problems (up to 400 variables) and for single solution *AIM* generated problems [Cha and Iwama, 1995]. To see if these results still hold, we updated Cha and Iwama's study by comparing our constraint weighting algorithms with McAllester *et al.*'s [1997] implementation of RNOVELTY, NOVELTY, BEST and TABU. For our problem set we randomly generated 100, 200 and 400 variable 3-SAT problems with a clause/variable ratio of 4.3. This placed the problems in the accepted phase transition area where the probability of satisfiability is approximately 0.5 [Mitchell *et al.*, 1992]. From this we selected the first ten satisfiable problems for each problem size (shown as *r100*, *r200* and *r400* in table 4.1). At each problem size we calculated the average of 100 runs. We also used the 4 *AIM* generated single solution 100 variable problems available from the DIMACS benchmark set (see ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf). Each problem was solved 100 times and the averages reported. Table 4.1 shows the results for these problems[1] and confirms constraint weighting's superiority for small *AIM* formula, but indicates NOVELTY and RNOVELTY have better random 3-SAT performance. The results also show there is a growing difference between constraint weighting and the other techniques as the problem size increases. For the *r400*

[1]All problems were solved on a Sun Creator 3D-2000

problems NOVELTY is still solving 100% of instances within 1,000,000 flips where the success rate for the best weighting strategy (MOVEWGT) has dropped to 61.5%. Also, for the larger problems, BEST starts to outperform the weighting algorithms. Of the weighting strategies, each has similar performance, although UTILWGT starts to do better and MOVEWGT worse as problem size increases. (In table 4.1, Cut-off is the number of flips at which *unsuccessful* runs were terminated, all Mean, Median, Max, Min and Std Dev statistics are for *successful* runs, and success is the percentage of problems solved before the cut-off number of flips).

Problem	Method	Flips		Time (seconds)					Success
		Mean	Cut-off	Mean	Median	Max	Min	Std Dev	
r100	NOVELTY	1331	250000	0.029	0.014	0.366	0.0009	0.045	100.0%
(100 vars	RNOVELTY	1454		0.032	0.012	0.528	0.0009	0.055	100.0%
432 cons)	TABU	1820		0.035	0.017	0.564	0.0009	0.053	100.0%
	MOVEWGT	1988		0.034	0.011	2.057	0.0007	0.116	100.0%
	MINWGT	2068		0.038	0.012	2.169	0.0007	0.123	100.0%
	UTILWGT	2670		0.042	0.010	3.528	0.0006	0.171	99.7%
	BEST	4072		0.077	0.030	1.120	0.0013	0.121	100.0%
r200	RNOVELTY	25422	500000	0.552	0.125	10.164	0.0022	1.387	96.9%
(200 vars	NOVELTY	29014		0.632	0.144	10.614	0.0029	1.379	98.9%
864 cons)	UTILWGT	43959		0.727	0.198	8.053	0.0029	1.295	89.3%
	MINWGT	45087		0.822	0.191	8.955	0.0024	1.545	89.9%
	BEST	46946		0.875	0.277	9.031	0.0047	1.441	98.7%
	MOVEWGT	50554		1.550	0.369	15.124	0.0037	2.739	85.6%
	TABU	57305		1.141	0.267	9.769	0.0053	1.909	91.5%
r400	RNOVELTY	85175	1000000	1.891	0.478	21.696	0.0112	3.294	94.7%
(400 vars	NOVELTY	108497		2.374	0.671	21.507	0.0117	3.804	93.9%
1728 cons)	BEST	147933		2.759	1.094	18.178	0.0173	3.823	92.5%
	UTILWGT	160702		2.722	1.108	16.613	0.0126	3.634	60.7%
	MINWGT	164093		3.278	1.305	19.910	0.0136	4.357	57.9%
	MOVEWGT	175473		5.516	2.242	30.917	0.0307	7.042	61.5%
	TABU	264821		4.993	2.793	18.757	0.0578	5.119	44.6%
AIM 100	MOVEWGT	4410	250000	0.084	0.041	3.066	0.0021	0.228	100%
(100 vars	MINWGT	4504		0.058	0.046	0.260	0.0034	0.043	100%
200 cons)	UTILWGT	10789		0.138	0.111	0.857	0.0044	0.115	100%
	RNOVELTY	-		-	-	-	-	-	0%
	NOVELTY	-		-	-	-	-	-	0%
	TABU	-		-	-	-	-	-	0%
	BEST	-		-	-	-	-	-	0%

Table 4.1. Results for small 3-SAT problems[2]

To investigate the gap between constraint weighting and WSAT for larger problems, we looked at the relative performance of the weighting algorithms with

[2] In this and succeeding result tables techniques have been ordered according to overall performance based on both percentage of problems solved and average number of moves (or flips)

RNOVELTY for the DIMACS benchmark large 3-SAT problems (800 to 6400 variables). The graph in figure 4.2 shows the best result obtained for each algorithm (after 10 runs of 4 million flips on each problem) and confirms that constraint weighting performance starts to degrade as problem size increases. However, an interesting effect is that the relative performance of UTILWGT continues to improve as the problem size grows until it significantly dominates the other weighting methods.

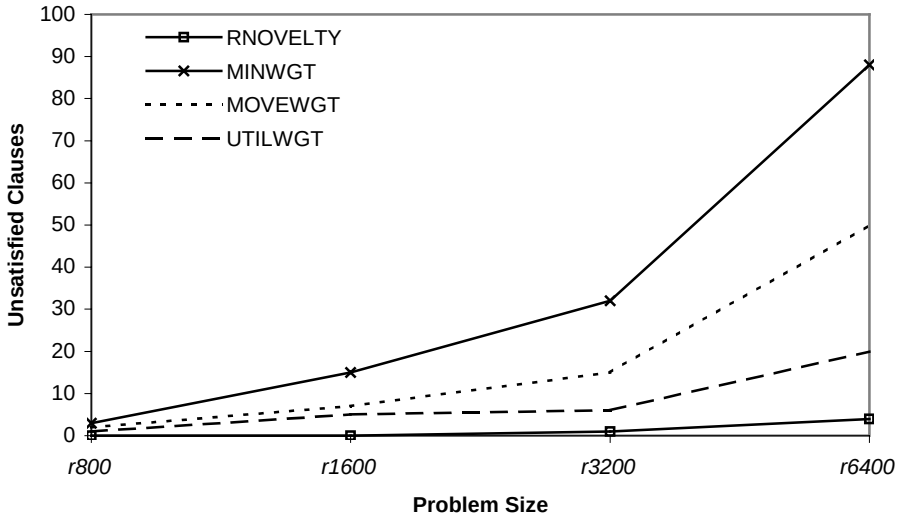


Fig. 4.2. Result plot for large DIMACS 3-SAT problems

To test whether the random 3-SAT results are reproduced in more structured domains we looked at a selection of conjunctive normal form (CNF) encodings of realistic problems again taken from the DIMACS benchmark. These problems comprised of two large graph colouring problems (g125.18 and g250.15), four inductive inference problems (*ii32*: ii32b3 to ii32e3), four circuit fault diagnosis problems (*ssa*: ssa7552-038 to ssa7552-160) and two parity function learning problems (*par*: par8-2-c and par8-4-c).

Results averaging 100 runs on each problem in each category are given in table 4.2, and show constraint weighting performed relatively poorly on the hard graph colouring problems but was superior to the WSAT techniques on the other DIMACS problems. No one weighting heuristic was superior in all cases: UTILWGT was better on inductive inference and graph colouring (supporting the earlier finding that the relative performance of UTILWGT improves as problem sizes becomes larger) and MOVEWGT was better on circuit diagnosis and parity learning.

However, given the results for the better methods on each problem class are very similar (considering the standard deviations in table 4.2), only large differences can be considered significant. Bearing this in mind, we cannot clearly distinguish between weighting and novelty on the parity problems or between the better weighting methods on the *ssa* and *ii32* problems.

Problem (max size)	Method	Flips		Time (seconds)					Success
		Mean	Cut-off	Mean	Median	Max	Min	Std Dev	
g125.18	NOVELTY	5915	1000000	1.639	1.494	5.086	0.3181	0.576	100.0%
g250.15	RNOVELTY	6880		1.741	1.598	5.397	0.2738	0.603	99.5%
graph	UTILWGT	21921		3.741	2.847	47.883	0.6830	3.026	100.0%
colouring	BEST	24566		3.609	3.052	27.352	0.5657	2.416	100.0%
(3750 vars	TABU	26025		4.885	4.158	25.091	0.6830	3.002	100.0%
233965 cons)	MINWGT	34240		5.338	4.700	32.315	0.5790	3.045	100.0%
	MOVEWGT	218168		33.037	29.435	145.909	1.5723	16.627	88.0%
ssa (038-160)	MOVEWGT	2885	250000	0.052	0.043	0.258	0.0211	0.027	100.0%
circuit	MINWGT	3085		0.081	0.070	0.503	0.0340	0.045	100.0%
fault	UTILWGT	11532		0.164	0.100	2.117	0.0218	0.199	99.8%
diagnosis	NOVELTY	26987		0.672	0.313	6.150	0.0332	0.990	97.3%
(1501 vars	RNOVELTY	29541		0.767	0.265	6.166	0.0385	1.259	94.0%
3575 cons)	BEST	29606		0.407	0.290	3.264	0.0461	0.373	99.8%
	TABU	58975		0.715	0.488	3.027	0.0517	0.638	87.0%
par (8-2c,8-4c)	MOVEWGT	2542	250000	0.052	0.026	0.867	0.0011	0.088	100.0%
parity	RNOVELTY	2760		0.049	0.038	0.242	0.0010	0.043	100.0%
function	NOVELTY	2796		0.050	0.034	0.226	0.0008	0.047	100.0%
learning	MINWGT	3098		0.046	0.019	0.623	0.0010	0.089	100.0%
(68 vars	UTILWGT	7750		0.106	0.020	2.948	0.0006	0.330	99.0%
270 cons)	BEST	25183		0.381	0.201	3.297	0.0016	0.480	100.0%
	TABU	27635		0.417	0.153	3.245	0.0015	0.630	99.5%
ii32 (b3-e3)	UTILWGT	916	250000	0.157	0.110	0.706	0.0325	0.128	100.0%
inductive	MINWGT	1156		0.219	0.109	2.356	0.0306	0.312	100.0%
inference	BEST	1185		0.122	0.076	1.412	0.0205	0.122	100.0%
(824 vars	MOVEWGT	2739		0.483	0.150	2.971	0.0272	0.663	100.0%
19478 cons)	TABU	4324		0.391	0.132	2.980	0.0120	0.534	100.0%
	RNOVELTY	18477		2.911	2.445	23.606	0.0325	2.517	100.0%
	NOVELTY	51391		8.054	5.343	37.856	0.0302	8.670	47.0%

Table 4.2. Results for structured DIMACS problems

The overall satisfiability results show that the WSAT techniques tend to dominate for very large problems and for randomly generated problems. Conversely, the constraint weighting algorithms do better on the smaller realistic problems and on the artificially generated *AIM* problems (the *AIM* problems were built up by starting with a solution and then generating a problem that is only satisfied by that solution [Asahiro *et al*., 1993]).

So far the results suggest that weighting performs less well in longer term searches: in the domains where constraint weighting dominates, (e.g. *AIM*, parity

learning and circuit fault diagnosis) all solutions are found relatively quickly. In the large, randomised *and* difficult problems (e.g. 3-SAT and graph colouring) the constraint weighting heuristics do not seem to provide effective long term guidance.

4.4.2 CSP Results

Satisfiability is a subset of the broader domain of constraint satisfaction. Although CNF formulations can model multiple problem domains, they all share the same constraint type (i.e. clauses of disjunct literals). For many CSPs there are more natural and efficient ways of modelling constraints and variables. It is therefore significant to explore the performance of our algorithms on a broader range of problems. To this end, we looked at two CSP formulations of real-world problems (university timetabling and nurse rostering), both involving complex non-binary constraints and large non-standard variable domains (see Chapter 3). In addition we ran tests on the well-studied problem of random binary constraint satisfaction [Prosser, 1996].

For the purpose of the research, a university timetable problem generator was developed. The generator can be tuned to produce a wide range of realistic problems, while also having a mode that creates relatively unstructured, randomised problems. We were interested in building identical sized problem pairs, one reflecting the structure of a realistic timetabling problem (i.e. students doing degrees, following predictable lines of study, etc.) and the other using purely random allocations. The motivation was to test if a realistic problem structure influences the relative performance of the algorithms.

The nurse rostering experiments were run on a set of benchmark problems, taken from a real hospital situation. Each schedule involves up to 30 nurses, over a 14 day period, with non-trivial constraints defining the actual conditions operating in the hospital (for more details, see Chapter 3 and [Thornton 1995]).

Finally, two sets of hard random binary CSPs were generated, with 30 variables of domain size 10, one with a constraint density of 80% and constraint tightness of 17% and the other one with a constraint density of 40% and constraint tightness of 32%. This placed the problems in the accepted phase transition area [Prosser, 1996] and made them large enough to challenge the standard backtracking techniques.

Problem (size)	Method	Mean Iterations	Cut-off	Time (seconds)					Success
				Mean	Median	Max	Min	Std Dev	
tt-struct	TABU	200174	10^6 iter	70.24	44.00	407.00	7.00	73.6523	82%
(500 cons)	NOVELTY	228503		72.58	48.42	331.04	8.04	75.7549	84%
	RNOVELTY	213110		62.90	47.83	259.79	9.29	51.0738	78%
	MINWGT	239914		85.76	47.04	452.51	12.81	95.3296	74%
	MOVEWGT	250283		105.84	48.00	694.00	11.00	131.1727	70%
	UTILWGT	330810		130.52	74.00	481.00	14.00	125.2034	62%
	BEST	245826		67.91	48.00	289.00	16.00	78.6848	11%
tt-rand	NOVELTY	106540	10^6 iter	28.95	23.36	109.71	12.55	18.5927	100%
(500 cons)	RNOVELTY	117566		31.65	23.14	203.78	11.65	29.9896	97%
	MINWGT	111360		29.86	26.86	66.51	16.54	10.2034	95%
	MOVEWGT	112814		33.56	25.00	294.00	15.00	31.5215	97%
	UTILWGT	134489		38.47	31.00	180.00	18.00	22.3877	95%
	TABU	98555		32.35	18.00	352.00	9.00	50.7071	91%
	BEST	409520		105.03	100.00	240.00	21.00	59.8682	36%
roster	MINWGT	125738	400 sec	54.10	24.01	395.51	2.24	75.2091	94%
(500 cons)	UTILWGT	135010		59.01	23.00	332.00	1.00	82.1876	80%
	MOVEWGT	202222		86.75	42.00	393.00	3.00	89.2773	80%
	BEST	649476		72.07	38.00	362.00	3.00	86.4546	84%
	TABU	501647		43.97	7.00	397.00	1.00	97.4967	67%
	NOVELTY	545464		53.29	12.00	282.00	1.00	72.2156	73%
	RNOVELTY	874954		72.40	18.77	356.78	2.07	101.4248	76%
bin80	TABU	80175	$2x10^6$ iter	0.945	0.55	5.78	0.03	1.0874	100%
n=30	RNOVELTY	103155		1.239	0.67	5.42	0.01	1.3382	100%
m=10	BEST	211860		2.786	2.03	11.06	0.07	2.6493	100%
p1=80	MOVEWGT	264698		2.922	0.60	21.55	0.04	4.9709	89%
p2=17	MINWGT	243895		2.950	0.95	23.10	0.07	5.1019	79%
(200 cons)	UTILWGT	242285		3.092	0.68	25.31	0.06	4.9914	75%
	NOVELTY	207728		2.391	1.07	16.76	0.02	3.4041	55%
	Backtrack	$2.4x10^9$	10^3 sec	408.600	n/a	n/a	n/a	n/a	80%
bin40	TABU	90124	$2x10^6$ iter	0.609	0.37	3.34	0.01	0.6729	100%
n=30	RNOVELTY	198933		1.361	0.58	9.79	0.01	1.8367	100%
m=10	BEST	288590		2.255	1.20	15.00	0.04	2.8831	100%
p1=40	MOVEWGT	134950		0.848	0.15	9.67	0.05	1.7390	79%
p2=32	UTILWGT	209704		1.476	0.31	13.32	0.02	2.5588	57%
(200 cons)	MINWGT	221497		1.465	0.31	11.11	0.04	2.3430	55%
	NOVELTY	200067		1.368	0.66	6.32	0.01	1.7171	48%
	Backtrack	33923486	10^3 sec	16.400	n/a	n/a	n/a	n/a	100%

Table 4.3. Results for CSPs (*tt-struct* = structured timetabling, *tt-rand* = random timetabling, *roster* = nurse rostering, *bin* = binary CSP)

Table 4.3 shows the results of running each class of problem against our existing algorithms (all results are averages of 100 runs = 10 runs × 10 different problems). We also report results for the binary CSPs using Van Beek's backtracking algorithm (see ftp://ftp.cs.ualberta.ca/pub/vanbeek/software). The table 4.3 results show little distinction between the best techniques for both classes of timetable problem, but favour TABU for the binary CSPs and MINWGT and UTILWGT for

the nurse rostering problems. Adding structure to the timetabling problems does slow performance, but does not seem to favour a particular method.

4.4.3 Constraint Weight Curves

To further investigate the behaviour of constraint weighting, we looked at the way constraint weights are built up during a search. To do this we developed *constraint weight curves* which plot the constraint weights on the y-axis and order the constraints on the x-axis according to their ascending weight values. For example, if at the solution point of a problem with 4 constraints *a*, *b*, *c* and *d*, constraint *a* has a weight of 2, constraint *b* has a weight of 4, constraint *c* has a weight of 1 and constraint *d* has a weight of 10, after normalising these weights on a 0-100 scale, we would produce the constraint weight graph shown in figure 4.3.

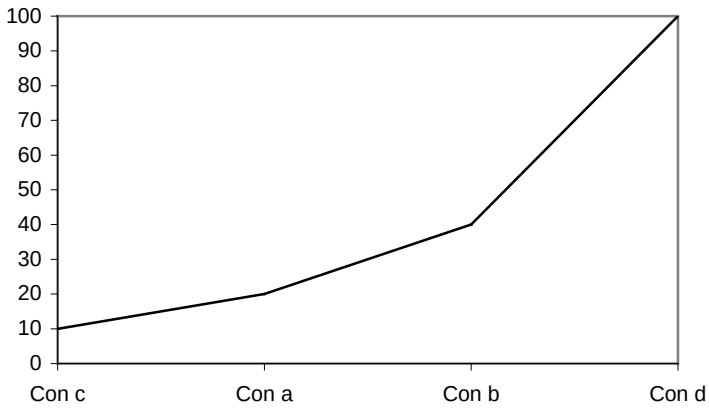


Fig. 4.3. An example constraint weight curve

Constraint weight curves provide a picture of the distribution of weights across the constraints. For example, Figure 4.4 shows the constraint weight curves for the *r100* to *r400* 3-SAT problems using the MOVEWGT heuristic (each curve is the average of ten runs on ten different problems again normalised on axes from 0 to 100):

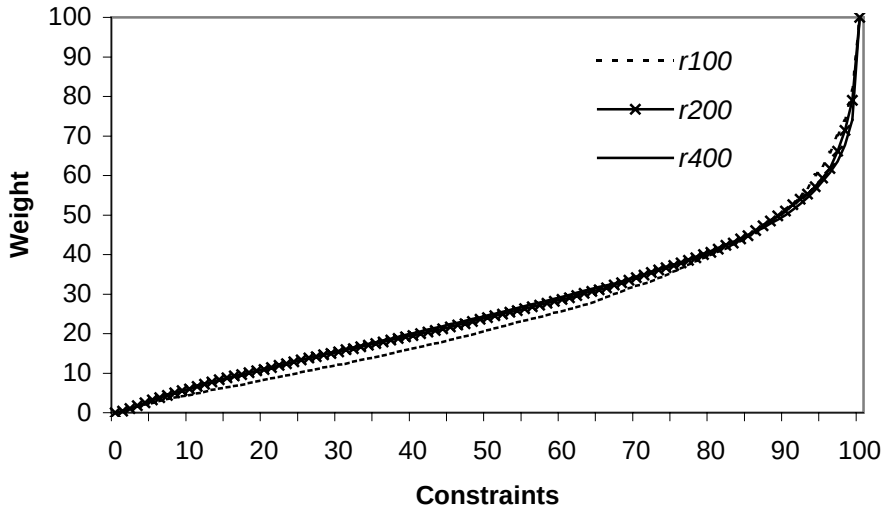


Fig. 4.4. Constraint weight curves for various 3-SAT problems

We also created curves for the larger DIMACS 3-SAT problems (800 to 6400 variables) and found that after an initial predictable adjustment period, curves very similar to those in figure 4.4 are produced. To see if this effect is consistent across weighting strategies we plotted the averaged curves for each weighting strategy solving a range of 200 to 6400 variable 3-SAT problems (shown in figure 4.5). In combination, these curves show a remarkable consistency between methods and across problem sizes and indicate that in the longer term, the weighting process mainly serves to smooth the curves closer to an underlying distribution.

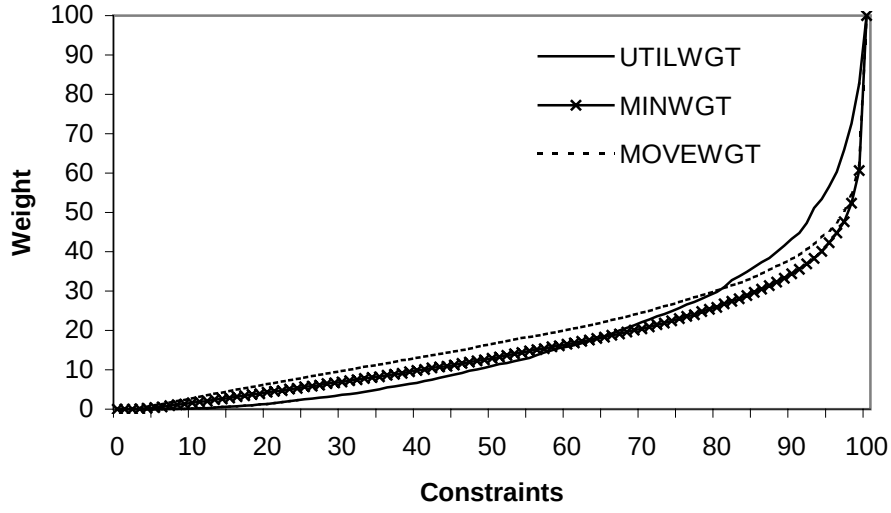


Fig. 4.5. Constraint weight curves for different constraint weight methods

We therefore became interested in finding a function that expresses this underlying distribution. After some trial and error, we found the best fit occurred with functions of the form $y = a - b\log_n(c - x)$. For example, figure 4.6 shows the match between one of the 3-SAT curves and $y = 87.5 - 19\log_n(101 - x)$.

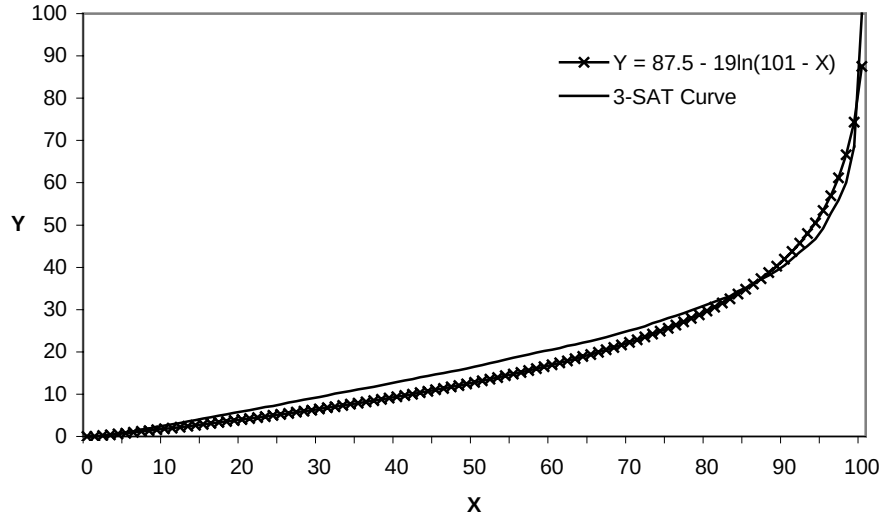


Fig. 4.6. 3-SAT and log function comparison

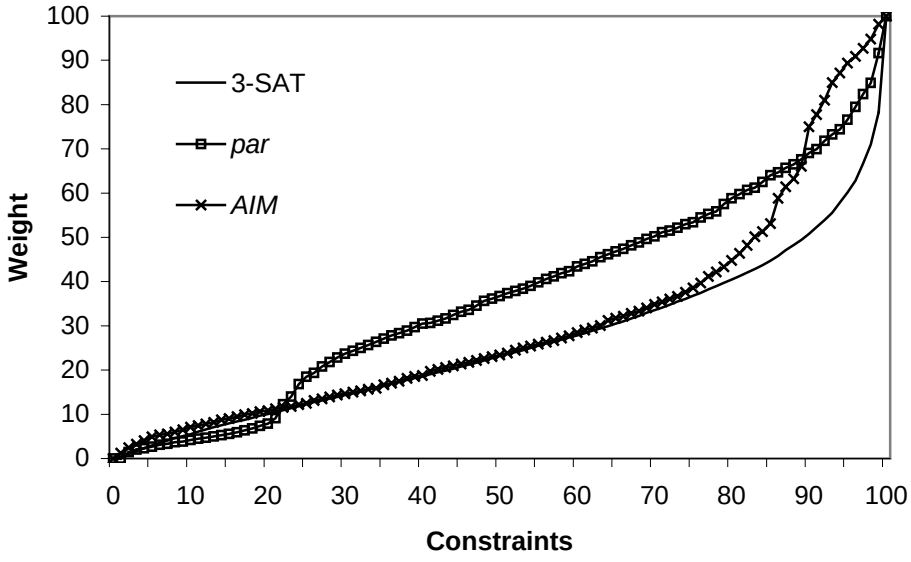


Fig. 4.7. Non-uniform DIMACS constraint weight curves

To further explore this phenomenon we looked at the constraint weight curves for the other CSP and DIMACS problems. For clarity, the *AIM* and parity learning curves are shown in figure 4.7, the other DIMACS curves are shown in figure 4.8 and the CSP curves are shown in figure 4.9. In all graphs the *r400* 3-SAT curve is given to enable comparison.

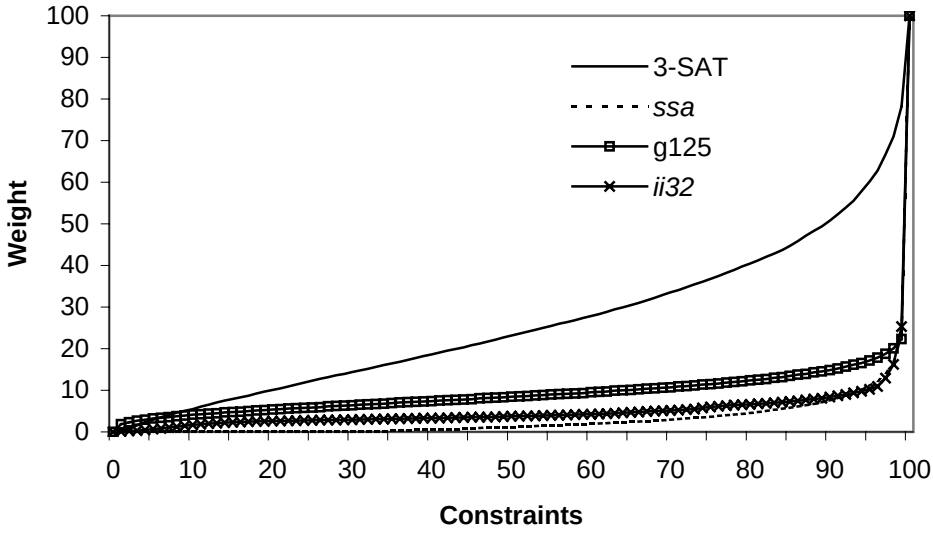


Fig. 4.8. Uniform DIMACS constraint weight curves

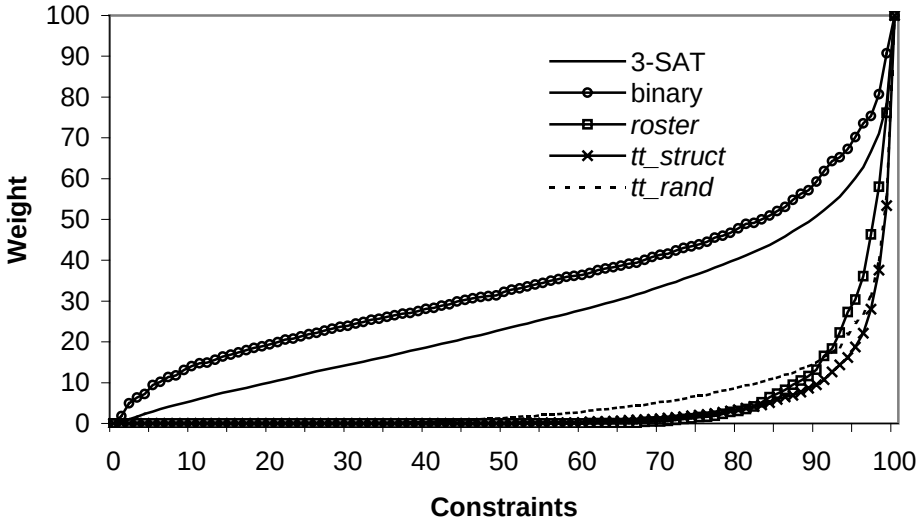


Fig. 4.9. CSP constraint weight curves

The first feature observed from these curves is that there is a high degree of consistency within problem domains but noticeable differences between domains. The three figures also show we obtained three types of constraint weight curve for the problems considered: the curves in figures 4.8 are similar (or uniform) in that after an initial steeper start all the curves have a steadily increasing gradient. These curves differ mainly in the steepness of their ascent and in the point at which the curve reaches the constraint axis. The curves in figure 4.7 for the *AIM* and *par* problems show different behaviour, in that each curve deviates from the steadily increasing gradients of the other problems and exhibits some irregularity. For instance, the *par* problems show a 'step' between 22 and 28 on the constraint axis and the *AIM* problems show a similar

step between 80 and 95. Finally the roster and timetabling CSP curves in figure 4.9, although showing a steadily increasing gradient do not show *any* weight accruing to the first 40% of constraints (for the *roster* and *tt_struct* curves over 60% of constraints are unweighted). The binary CSP curve however shows noticeable similarity to the original 3-SAT curve.

4.4.4 Constraint Trajectories

As the constraint weight curves for all problems were found to settle into fairly fixed distributions, we became interested in testing whether the individual constraint weights also converged to a fixed order. To do this we plotted the relative positions of single constraints, within the total order of constraints, at different points during the search. Figure 4.10 shows example distributions for 4 constraints (clauses) taken from a 6400 variable 3-SAT problem. To understand this graph, consider constraint *B* (represented by the single continuous line): this line ends with a value of 94 on the Order axis at 8 million iterations, meaning (on a scale of 0 to 100) constraint *B* was the 94^{th} most heavily weighted at 8 million iterations (note, as there are 27648 constraints in the problem, 275 other constraints will also be categorised in 94^{th} position). Although the graph shows significant variation in the relative positions of each constraint, a regular pattern of peaks and troughs does occur. This pattern can be understood as representing a constraint changing from true to false and false to true. For example, again considering *B*, this constraint starts in a high position (99) indicating it was initially false and so was weighted at the beginning of the search. After this *B* becomes true and so does not accrue any more weight. As other constraints are being weighted this causes the relative position of *B* to decline until it reaches such a point (at 800,000 iterations) that the search decides to make it false (in effect it has insufficient weight to remain true). After this *B* remains false and so accrues weight at each local minimum, causing the steep ascent from 800,000 to 900,000 iterations. At this point it has accrued enough weight for the search to make it true again and so it starts a second decline as other false constraints accrue more weight. Understood in this way, the graph in figure 4.9 suggests that behind the peaks and troughs the overall importance (or difficulty) of a constraint is best measured by taking the average of the peak values in the constraint's trajectory. In effect, the height of a peak shows how hard it is

for a constraint to become true in the current local neighbourhood and successive peaks measure different neighbourhoods. From this we can conclude that, without knowing the trajectory of a constraint, the weight on a constraint at a particular point in the search does not tell us much about the importance of that constraint (unless the weight is relatively very high or very low).

To confirm our results we looked at the trajectories of randomly selected constraints from each of our other problem domains. In most cases the same pattern of sharp peaks with shallower declines was observed, with the exception of certain constraints that accrued very high or very little weight and maintained fairly straight trajectories (this is examined in the next section). We also observed that in the domains where problems are solved relatively quickly in relation to the number of constraints (e.g. circuit diagnosis and inductive inference) the majority of constraints only achieved a single peak before a solution was found (i.e. became true and never became false again).

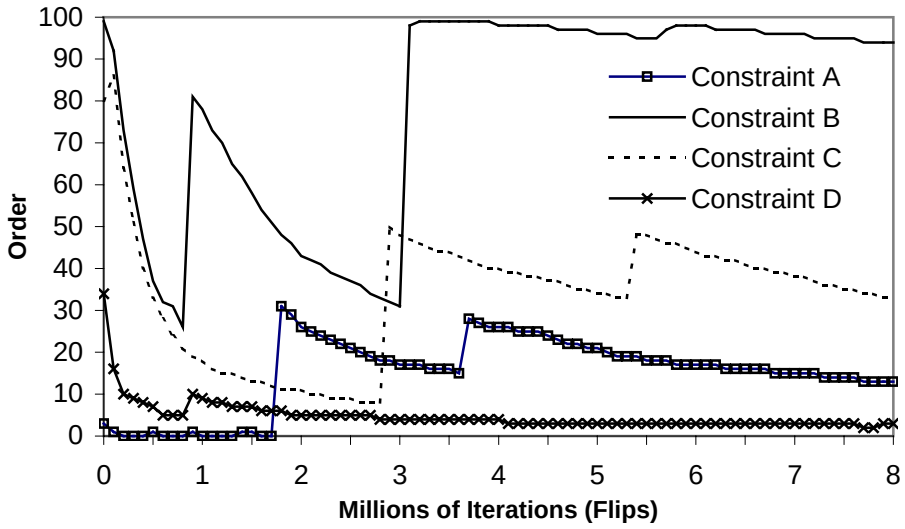


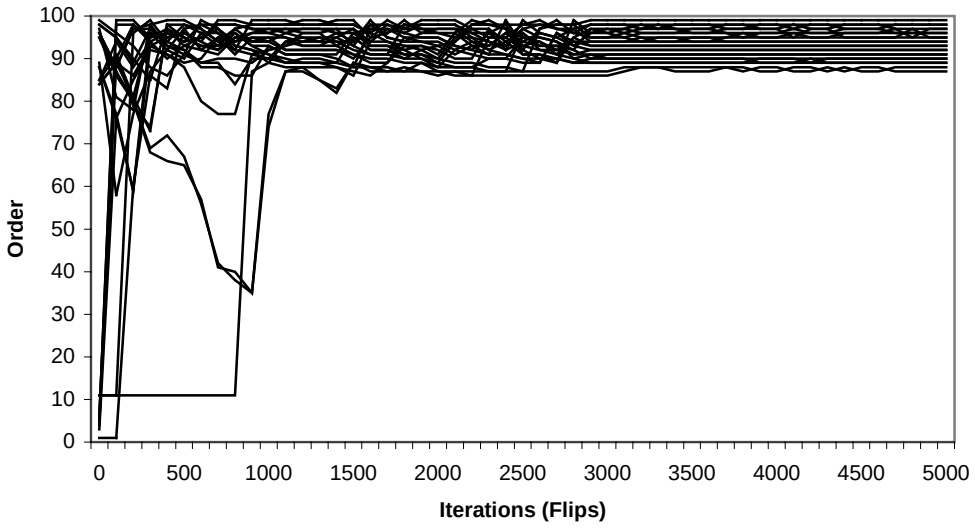
Fig. 4.10. Changing weight order for 4 selected constraints (DIMACS 6400 3-SAT problem)

***AIM* Trajectories.** Of all the problem domains, the constraint weighting algorithms most clearly dominated on the *AIM* problems. We therefore decided to investigate the *AIM* constraint trajectories in more detail to see if there are any general features that cause the improved performance.

In contrast to the random problems, the *AIM* constraint weight curve exhibits a bulge or a step starting around value 85 on the constraint axis (see Figure 4.7). This suggests the constraint weight algorithm has found a group of especially dif-

ficult constraints, distinct from the rest of the problem, and is able to exploit this difference to solve the problem more efficiently. This exploitation would take the form of ensuring the constraints in this group are kept *simultaneously* true (by frequently placing weights on them) and then exploring the search space by violating constraints in the easier constraint group. In doing this, constraint weighting will fix potential bottlenecks early in the search and quickly move to potential solution areas (this is discussed further in section 4.5). In contrast, non-weighting methods do not distinguish moves that violate difficult constraint groups and so are more likely to move into constraint violations that are harder to repair.

Fig. 4.11. Weight trajectories of the 24 most heavily weighted *AIM* 1 constraints



To test this hypothesis, we plotted the trajectories of the most heavily weighted constraints in each of the *AIM* and *r100* problems. These graphs showed a distinct pattern emerging for the *AIM* problems with groups of constraints having parallel trajectories. For example, figure 4.11 shows the trajectories for the first 24 most heavily weighted constraints in the DIMACS aim-100-2-0-yes-1.cnf problem, figure 4.12 shows the trajectories for the next 10 most heavily weighted constraints from the same problem and figure 4.13 shows the trajectories for the 24 most heavily weighted constraints in an example *r100* problem. The *AIM* graphs show a definite split occurs between the 24^{th} and 25^{th} constraints: the first group quickly converge to their relative positions and then maintain a high weight order for the rest of the search, while the remaining constraints in figure 4.12 follow the more typical ‘peak and trough’ behaviour shown by the *r100* graph (figure 4.13) and de-

scribed in the previous section. The *AIM* constraint division can be explained by the weighting algorithm *continually* adding weights to the first 24 constraints (because satisfying one constraint in this group quickly causes another to become false) and supports the reasoning that the 'bulge' in the *AIM* constraint weight curve is caused by groups of constraints that are difficult to satisfy.

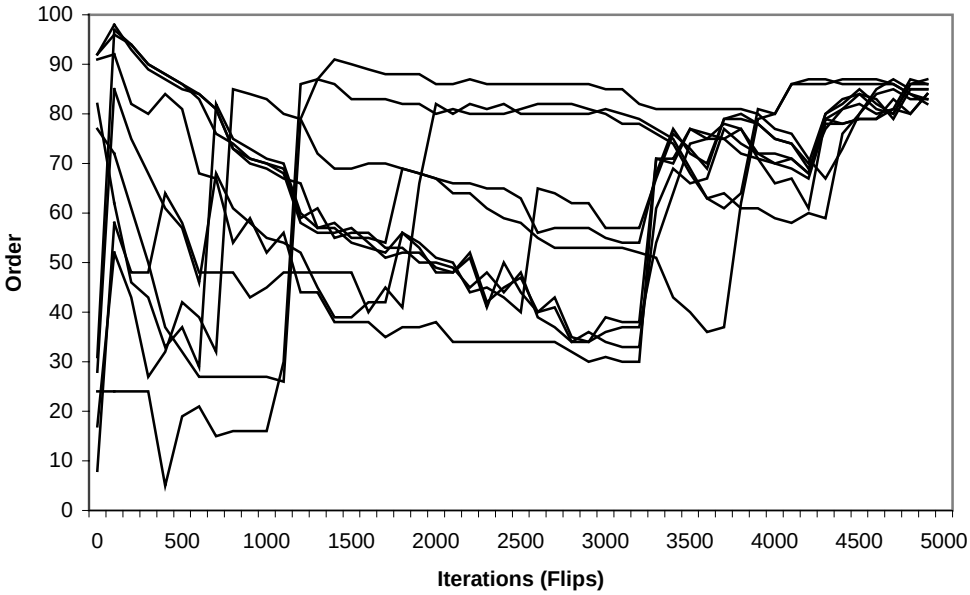


Fig. 4.12. Weight trajectories of the second 10 most heavily weighted *AIM* 1 constraints

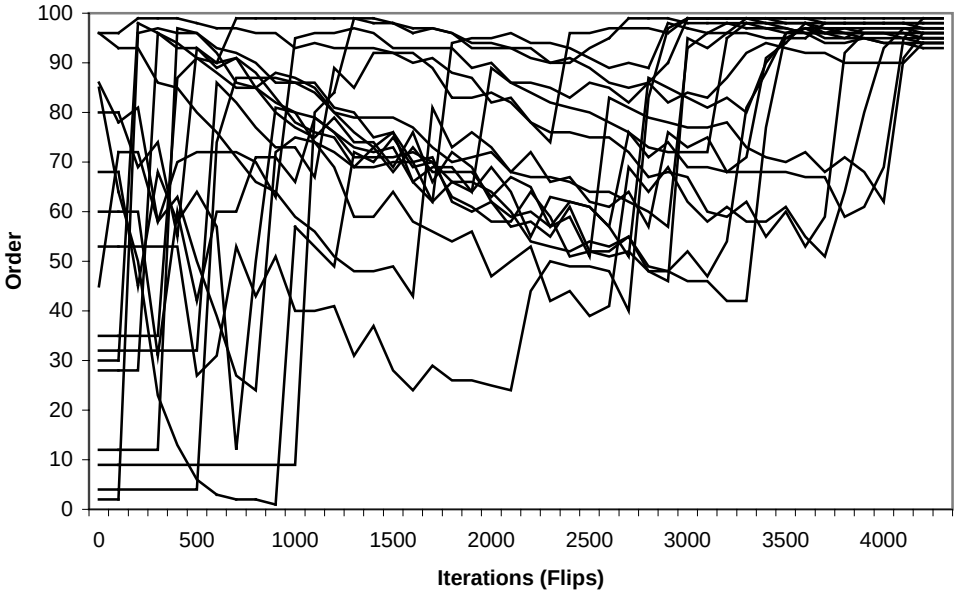


Fig. 4.13. Weight trajectories of the first 17 most heavily weighted *r100* constraints

4.4.5 Measuring Constancy

The *AIM* problem results suggest constraint weighting does better when it can recognise and simultaneously satisfy a group of difficult constraints. The parallel *AIM* trajectories imply that if there is little change in the ordering of the heaviest constraints during a search then constraint weighting has recognised such a group of difficult constraints and is therefore likely to have some advantage over non-weighting methods. To test this hypothesis we developed a constancy measure *Ct* that compares the membership of H_{half} (the set of the top 10% of heaviest weighted constraints halfway through a search) with H_{full} (the set of the top 10% of heaviest weighted constraints at the end of the search). Specifically:

$$Ct = | H_{half} \cap H_{full} | \div | H_{half} |$$

i.e. we measure the proportion of constraints that are in the top 10% at the halfway point that are still in the top 10% at the end of the search. *Ct* therefore varies from zero to one, where zero represents no member of H_{half} being in H_{full} and one represents all members of H_{half} being in H_{full}. We developed *Ct* measures for each of our problem domains based on an average of at least 100 runs with each run stopping at the appropriate median iteration level reported in tables 4.1, 4.2 and 4.3. The results are reported in figure 4.14, with the exception of the non-binary CSPs (timetabling and rostering). These problems were not considered comparable on the *Ct* measure because only relatively few constraints receive any weight during the search (as shown in figure 4.9).

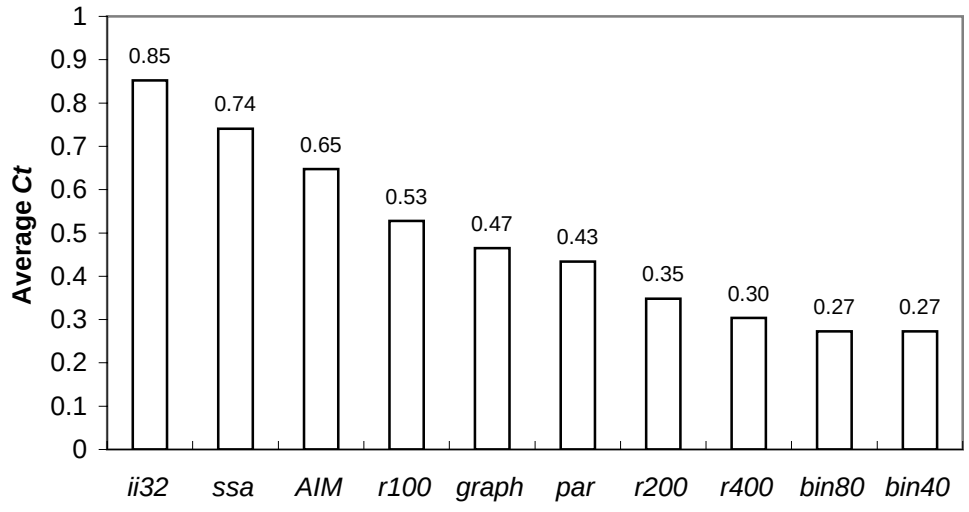


Figure 4.14. Constancy measure *Ct* of the top 10% of the heaviest weighted constraints

The figure 4.14 plots correspond well with the performance results of sections 4.4.1 and 4.4.2 with the problems where weighting does well (*ii32*, *ssa* and *AIM*) also having a high *Ct* measure. This bears out our reasoning that constraint weighting does better when it can find and consistently satisfy a group of more difficult constraints.

4.4.6 Measuring Problem Structure

Small World Measures. So far in our discussion we have assumed a problem to be structured if it is based on a realistic situation and random if it is constructed using some form of random allocation of variables to constraints. Recent studies [Watts and Strogatz, 1998; Walsh, 1999] have more formally classified problems as random or structured based on the graph topology formed by representing problem variables as nodes and problem constraints as edges [Gent *et al*. 1999]. The relative structure or randomness of a graph is then measured using a clustering coefficient C and characteristic path length L. C is defined as the average clustering of all nodes in a graph, where the clustering of an individual node n is the proportion of edges existing between the k neighbours of n in relation to the total number of possible edges ($k(k-1)/2$). L is then defined as the average of the shortest path lengths between all pairs of nodes. Walsh [1999] further introduces a proximity ratio μ, defined as the ratio of C/L normalised by C_{rand}/L_{rand} where C_{rand} and L_{rand} are the clustering coefficient and characteristic path length for a random graph with the same number of nodes and edges as the original graph. Using this ratio, Walsh defines a random graph as having a μ of one, a structured graph as having a μ of less than one and a "small world" graph as having a μ of greater than one. The assumption here is that a random graph will have little clustering with relatively short paths between nodes and that a structured graph (e.g. a lattice) will have high clustering with relatively long paths between nodes. The small world graph sits in the middle with high clustering *and* short path lengths. Walsh argues that many realistic problems exhibit some underlying structure but also have an element of randomness and so would be expected to have a small world topology.

Problem	L	L_{rand}	C	C_{rand}	μ
r100	1.7788	1.7740	0.2711	0.2297	1.1767
r200	1.9397	1.9232	0.1656	0.1218	1.3479
r400	2.1536	2.1341	0.1069	0.0626	1.6923
AIM 100	2.4226	2.3100	0.2565	0.0930	2.6301
graph colouring	2.4295	1.9963	0.3824	0.0362	8.6908
ssa	7.3983	5.5553	0.5883	0.1142	3.8689
par	2.7512	2.7279	0.3496	0.1023	3.3885
ii32	2.3625	1.8286	0.7862	0.1726	3.5261
tt_struct	1.5778	1.5808	0.7522	0.4191	1.7985
tt_rand	1.1596	1.1665	0.8493	0.8342	1.0241
roster	1.8907	1.9238	0.5044	0.1184	4.3344
bin80 (80,17)	1.2000	1.2000	0.8014	0.8000	1.0051
bin40 (40,32)	1.6043	1.6000	0.4077	0.4000	1.0165

Table 4.4. Averaged small world measures for each problem set

One of the concerns of the current study is to examine whether randomness or structure in a problem has any effect on the relative performance of the constraint weighting algorithms. We therefore calculated C, L, C_{rand}, L_{rand} and μ for various problems used in the study (averages of these measures are shown for each problem class in table 4.4). These results show that all problems either exhibit a random topology (binary CSPs, and random timetable problems) or some degree of small world topology. Interestingly, the small random 3-SAT problems have a nearly random topology but as problem size increases a stronger and stronger small world topology emerges (due to greater than random clustering). In addition, adding realistic assumptions to the timetabling problems does produce a small world structure (again due to greater than random clustering). However, μ does not seem to be an obvious predictor of constraint weighting performance: those problems where weighting does well (*ssa*, *ii32*, *AIM* and rostering) cannot be clearly distinguished from the other problems on the basis of μ alone. There is some indication that weighting benefits from a greater than random path length. For instance, the two problems where L most exceeds L_{rand} (*ssa* and *ii32*) are also problems that weighting found easy to solve. However this is less clear for the *AIM* results, where weighting does well but L and L_{rand} are similar, or for graph colouring, where weighting does poorly but L significantly exceeds L_{rand}. The clearest observation that can be made from the small world results is that constraint weighting does less well on purely random problems. This suggests that weighting relies on finding some structure within a problem to obtain leverage over the other methods, but it does not appear that this structure can be clearly identified using the

small world measures. For this reason we decided to look more carefully at the different ways variables are connected in the various problem domains.

Neighbour count measures. Given the small world measures categorised our problem set as either small world or random, we became interested in measuring the degree of structure *within* our small world problems. The idea was to recognise structure by looking for neighbour relationships that are distinctly different from a randomly generated problem. To do this we looked at the *distribution* of neighbour relationships between variables. Firstly, for each of our problems we generated at least 100 random graphs with the same number of variables and edges as the corresponding original problem. We then counted the number of neighbours for each node and measured (for each graph) the maximum, minimum, mean, median, standard deviation, skewness and kurtosis of the neighbour counts. We repeated the procedure for the graphs representing the original problems and calculated the average measures for each problem class (these results are summarised in table 4.5).

Problem	Max-Min Range	Mean	Median	Std Dev	Skewness	Kurtosis
r100	28.20	22.74	24.10	5.5779	0.0896	-0.0408
random	20.74		23.40	4.1533	0.1221	-0.0041
r200	34.50	24.25	26.00	6.4354	0.2267	-0.0425
random	25.06		25.01	4.5957	0.1539	0.0062
r400	41.50	24.96	26.60	6.6239	0.1853	0.1048
random	28.47		25.89	4.8298	0.1788	0.0264
AIM 100	5.50	9.10	9.00	1.1787	-0.0702	0.1890
random	14.10		9.90	2.8507	0.2650	0.0410
graph colouring	36.00	108.95	108.50	6.4994	0.2655	-0.1555
random	71.90		109.75	10.1620	0.0917	-0.0120
ssa	135.00	3.79	5.00	4.9305	19.1745	472.8120
random	10.73		4.42	1.9063	0.6157	0.2432
par	17.00	5.10	4.00	3.4003	3.0011	8.1012
random	9.65		5.95	2.1370	0.3850	0.0439
ii32	82.00	64.57	94.25	30.1816	-0.0493	-1.5454
random	43.30		65.30	7.2918	0.0844	0.0039
tt_struct	234.70	138.85	180.80	57.8651	-0.5566	-0.7396
random	51.31		139.42	8.8190	0.0200	-0.0023
tt_rand	50.00	277.00	277.90	10.1426	-0.4297	-0.2305
random	31.48		277.37	5.4869	-0.1944	0.0340
roster	4.70	14.72	15.10	1.3417	-0.5542	-0.0875
random	20.27		15.48	3.6889	0.2277	0.0412
bin80 (80,17)	8.50	23.20	23.20	2.1540	-0.4061	-0.0567
random	8.49		23.50	2.1540	-0.2315	-0.0111
bin40 (40,32)	10.20	11.60	11.40	2.5733	0.2835	-0.1178
random	10.47		12.13	2.5720	0.0604	-0.0463

Table 4.5. Statistics for variable neighbour counts by problem domain

As expected, we found the random graphs to have a roughly normal distribution of neighbour counts (the normal distribution having a zero skewness and kurtosis [Tabachnick and Fidell, 1989]). However, of our problem set, only *ssa* and *par* showed a distinctly non-normal distribution. A further examination of these problems showed that the non-normality was caused by certain regular structures of clauses that created a large set of variables in each problem with identical neighbour counts. We therefore looked further at the max-min range and standard deviation of neighbour counts to find more evidence of structure within the other problems. The graphs in figures 4.15 and 4.16 show the difference between original problem and random problem max-min ranges and standard deviations as a proportion of the random values. Here a negative value indicates the original value was less than the random value and vice versa.

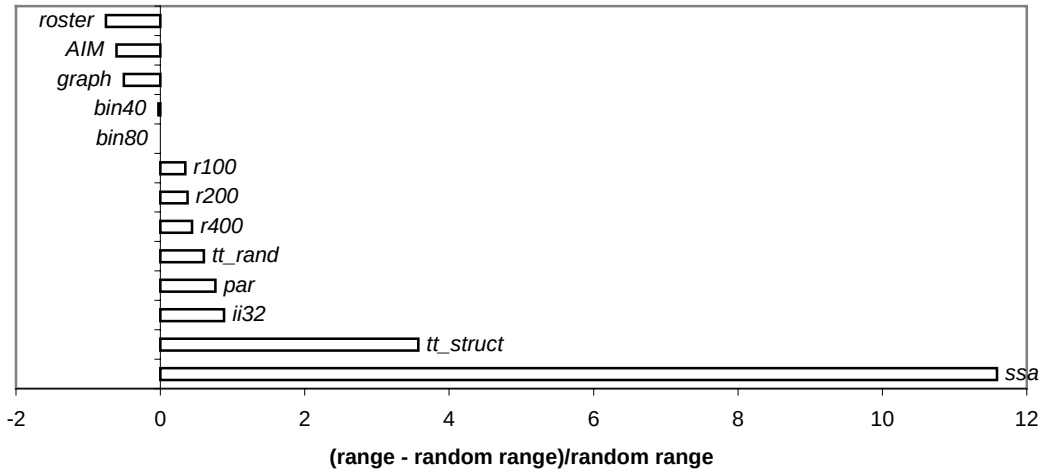


Figure 4.15. Neighbour count ranges as a proportion of random neighbour counts

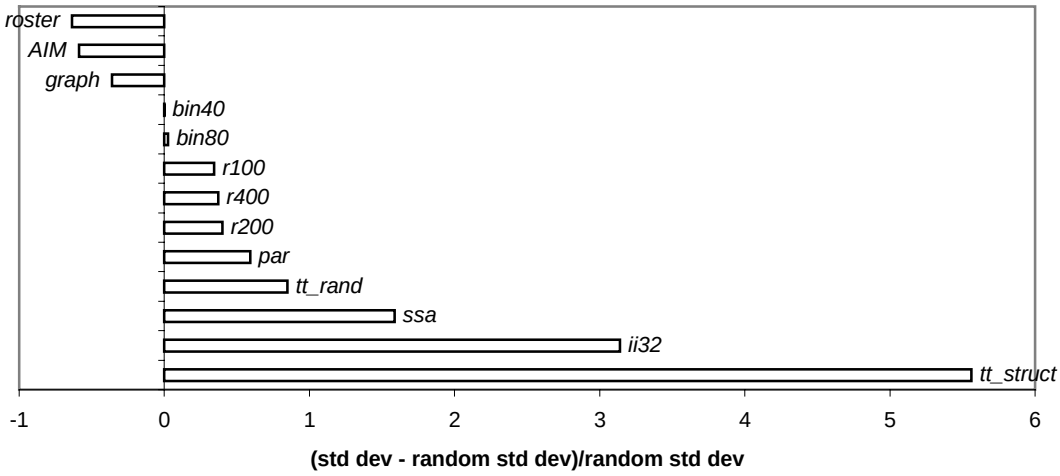


Figure 4.16. Neighbour count standard deviations as a proportion of random neighbour counts

As would be expected, the binary CSPs conform closely to their random counterparts, but all other problems (including random 3-SAT) show some degree of deviation from pure randomness. Given that the 3-SAT problems are randomly generated, we assume that the small increase in range and standard deviation over random for these problems is caused by the method of constraint representation (i.e. disjunct clauses of 3 literals). However the more significant positive deviations (for *ssa*, *tt_struct* and *ii32*) do seem to indicate problem structure, as do the negative deviations for graph colouring, *AIM* and *roster*. Taken together, the neighbour count measures reflect the level of structure in each problem that we would expect from our original knowledge of the problem domains. In addition they show the graph colouring, *AIM* and *roster* problems to be related in having less neighbours than would be expected and *ssa*, *tt_struct* and *ii32* to be related in having more neighbours than expected. Comparing the neighbour count results to the performance results for constraint weighting further supports the finding that weighting does better on more structured problems (or conversely, problems where weighting does worse (3-SAT, graph colouring and binary CSP) are also those problems with the closest to random neighbour distributions).

4.5 Analysis

4.5.1 Constraint Weighting Behaviour

When examining the behaviour of constraint weighting it is tempting to think in terms of ‘difficult’ and ‘easy’ constraints. However, in isolation (assuming a problem is not over-constrained) any constraint is easy to satisfy. It is the effect that satisfying a constraint has on the other constraints in a system that is significant. An easy constraint is one whose satisfaction has little effect on the rest of the problem, whereas a difficult constraint is one whose satisfaction tends to cause other constraints to become unsatisfied. However, difficult and easy constraints may *together* form a difficult constraint group (i.e. one that is difficult to satisfy entirely). A standard local search will tend to satisfy difficult individual constraints through cost guidance alone but is unable to recognise situations where the same constraints are interacting with each other and repeatedly becoming unsatisfied. In problems where such difficult sub-groups of constraints exist we would expect constraint weighting to do well, because it can recognise and penalise frequently violated constraints. In this way the whole group of con-

nected constraints will accrue weight, increasing the chances that the group is satisfied and moving the search to areas where there is more freedom of movement.

The existence of difficult constraint groups is a common feature in realistic problem solving. For instance, in nurse rostering areas of difficulty tend to focus on particular days where there is a staff shortage and so only involve constraints that are connected to that day. Similarly, timetabling problems generally have difficulty in scheduling classes in rooms where there is a high demand (e.g. computer labs and larger lecture theatres). Both these problems have examples of resource bottlenecks that only involve a limited set of constraints. Human problem solvers intuitively understand bottlenecks and tend to fix them first and then go on to solve the rest of the problem where allocations are less constrained. However, a local search without guidance will tend to keep revisiting a bottleneck because it is unable to recognise all the constraints involved. It is in this situation that constraint weighting is likely to outperform other non-weighting methods.

4.5.2 Identifying Hard Constraint Groups

Random problems. Our previous analysis of randomly generated problems has shown that we expect an approximately normal distribution of connections via constraints between variables. In such problems difficult constraint groups have to be generated by chance alone and are not part of the underlying problem structure. The 3-SAT and binary CSP constraint curves in figure 4.9 show that on these hard random problems, weights become spread across nearly all the problem constraints. The curves are also very similar in shape, exhibiting (after an initially steeper start) a constantly increasing gradient. In addition the random problems exhibit the lowest *Ct* measures (see figure 4.14), meaning there is a greater fluctuation in the membership of the heaviest weighted constraints. Combining this information suggests that although some constraints consistently accrue more weight than others, there is no *separation point* where a weighting algorithm can recognise that one constraint group is significantly different from another (reflected in the smoothly increasing slope of the 3-SAT and binary CSP constraint weight curves). This lack of distinction between constraint groups seems inherent in our

randomly generated problems and provides an explanation for constraint weighting's poorer performance on these domains.

Structured satisfiability problems. In contrast, for those problems where constraint weighting does well we have several indications that the weighting process has identified some non-random structure which it is able to exploit. For instance, the *ssa* and *ii32* constraint curves both exhibit a sharp turn near the end of their plots in figure 4.9. Taken in conjunction with the high *Ct* levels for these problems (between 0.74 and 0.85 in figure 4.14), a separate hard constraint group is indicated. Although the graph colouring plot also exhibits a sharp turn in figure 4.9, the *Ct* level for these problems is lower (0.47) meaning there is more fluctuation in membership in the peak group of constraints. Additionally the poorer performance of constraint weighting on graph colouring may be explained by the size of the problems (they were the largest in the set) rather than a lack of structure (both the constraint curves and the neighbour counts indicate some structure is present).

In the other satisfiability problems where constraint weighting does well (*AIM* and *par*) we again see the constraint curves deviate from the smoothly increasing random plots (see figure 4.7). In this case the *AIM* curve has the stronger indication of a separate hard constraint group (in the bulge at the top of the plot) which is further confirmed by the higher *AIM Ct* measure (0.65 versus 0.43 for *par*) and the superior performance of *AIM* (for *par* weighting is roughly equivalent to RNOVELTY). In fact, for the *par* problems, although there is an indication of structure from the neighbour count skewness and kurtosis measures, the constraint curve only shows a deviation from random in the early part of the plot (in figure 4.7), and the *Ct* and other neighbourhood measures are similar to the random results. Overall, this suggests that although there is some structure in the *par* problems this has not resulted in the delineation of a separate hard constraint group.

Structured CSPs. The realistic CSPs (rostering and timetabling) present a different interpretation problem. Here the weight becomes concentrated on a small group of constraints, with zero weight accruing on the remainder (the roster and timetable curves from figure 4.9 are reproduced in more detail in figure 4.17). For this reason the *Ct* measures were not interesting as membership of the top 10% of

constraints was fairly constant. Zero weight constraints are unlikely to provide leverage to a constraint weighting algorithm, because such constraints are *hard to violate* (instead of hard to satisfy). This means both weighting and non-weighting algorithms will tend to search in the space where these constraints are satisfied, although it is still possible for weighting to obtain leverage by finding a harder sub-group within the weighted constraints.

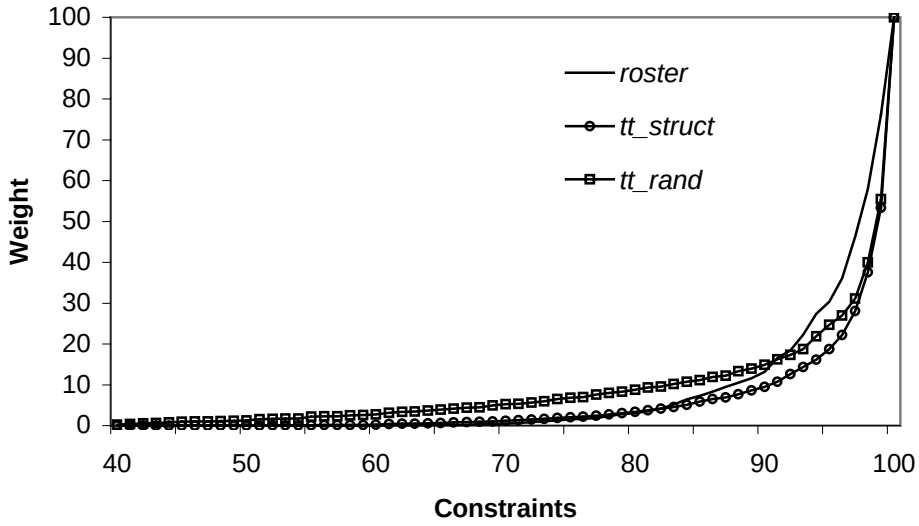


Fig. 4.17. Roster and timetabling constraint weight curves

Looking at the curves in figure 4.17, the weighted ranges for timetabling and rostering have smoothly increasing gradients and show no obvious separation of constraint groups. However the rostering curve is noticeably wider for the top 8% of constraints providing a possible alternative indication of a harder constraint group. It is significant to note that a similar effect also appears in the 'bulge' of the *AIM* curve of figure 4.7, where *AIM* initially follows the 3-SAT trajectory and then becomes wider at the end. The *AIM* and rostering problem structure measures are also similar, with both having less than random neighbour count ranges and standard deviations (see figures 4.15 and 4.16). Given that, of all problem domains considered, constraint weighting has the strongest performance advantage for the rostering and *AIM* problems (see tables 4.1 and 4.3) the similarities in constraint weight curves for these problems may be indicators of a hard constraint group. However, there is little *qualitative* difference between the roster and timetable curves and the *tt_rand* curve also diverges and is fatter than the *tt_struct* curve (at least until the top 2-3% of constraints) without a corresponding improvement in

weighting performance. We therefore looked in more detail at the weight distributions for higher end of the timetabling and rostering constraint weight curves. This is shown in figure 4.18 where we plot the top 5% most heavily weighted constraints for each problem.

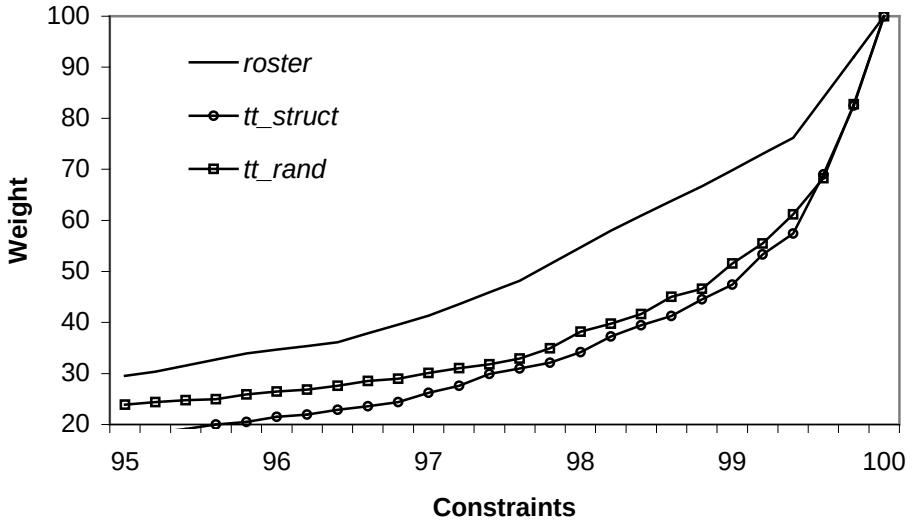


Fig. 4.18. Top 5% of roster and timetabling constraint weight curves

The figure 4.18 curves accentuate the differences between the timetabling and roster curves and show the roster curve maintains a greater width *throughout* the upper part of the graph. Given that roster problems tend to have bottlenecks on certain shifts which only involve 3 or 4 constraints, the greater top end width of the roster curve suggests weighting is penalising these relatively *small* hard constraint groups. As previously discussed (in section 4.5.1) recognising and simultaneously satisfying the constraints for a difficult shift would give weighting an advantage over non-weighting methods that tend cycle through violations of the same constraints.

In contrast, the timetabling curves start to meet and have a steeper gradient at the top end of the graph. This suggests weighting's ability to distinguish hard constraint groups is roughly equivalent for the structured and random timetabling problems and worse than for rostering (findings that correspond to the performance results of table 4.3).

Overall, the roster and timetable analysis suggests that the behaviour of constraint weight curves for the final 5-10% of constraints can be equally as signifi-

cant as the overall shape of the curve, as it is here that relatively small hard constraint groups can be recognised. It is of further interest to note that the UTILWGT curve in figure 4.5 is also slightly wider than the other weighting methods at the top end of the graph. This suggests UTILWGT is better able to distinguish relatively small hard constraint groups for the larger 3-SAT problems and provides an additional explanation of UTILWGT's superior performance on these problems. Finally it should be noted that the curve width arguments do not apply to the 3-SAT and binary CSP curves (in figure 4.9) because these curves, while being wider at the top end, remain wider throughout their length because of an *overall* lack of distinction between constraints.

4.5.3 Scaling Effects

Given that constraint weighting does better when it can find distinctions between groups of constraints, we would expect the probability of randomly generating hard constraint groups to decline (causing the performance of constraint weighting to also decline) as 3-SAT problem size increases. This is because the number of constraints in a random 3-SAT problem grows at the rate of $4.3n$ (where n is the number of variables) whereas the number of possible constraints grows at a faster rate of $2n(n\text{-}1)(n\text{-}2)$. Hence the probability of obtaining a particular pattern of constraint connections should decline as n increases. This reasoning is confirmed by looking at the mean neighbour count statistics in table 4.5 and the average neighbour clustering statistics (C) in table 4.4. Here the mean number of neighbours remains fairly static from n = 100 to n = 400 (moving from 22.74 to 24.96) but the average clustering starts to decline significantly (from 27.1% to 10.7%). Also the Ct measure starts to decline as random 3-SAT problem size increases (from 0.5275 to 0.3040 in figure 4.14) indicating there is less and less consistency in the top 10% of weighted constraints.

In addition there may be a *granularity* effect in constraint weighting, i.e. as the number of problem constraints increase, the effect of weighting a single constraint necessarily decreases, and constraints start to become weighted and violated in larger and larger groups. In this way the weight guidance becomes more general and less detailed, which could then cause promising search areas to be ignored. This is further backed up by the relative improvement in the performance of UTILWGT for longer

searches – as UTILWGT increments weights less frequently than the other methods we would expect it's performance to deteriorate more slowly.

4.5.4 Overall Behaviour

Performance. Overall, constraint weighting has done better on the *AIM*, *par*, *ssa*, *ii32* and nurse rostering problems. For each of these domains the constraint weight curves have deviated in some way from the smoothly increasing curves of the random 3-SAT and binary CSP problems (idealised in the log function of figure 4.6), indicating weighting has gained an advantage through being able to distinguish between constraint groups. Results for larger random 3-SAT and graph colouring problems further indicate that weighting gives poorer guidance as problem sizes grow. Of the non-weighting techniques, NOVELTY and RNOVELTY performed best on the 3-SAT, graph colouring and timetabling problems, while TABU dominated the binary CSP problems.

A comparison of weighting strategies does not favour one strategy in all situations. MOVEWGT performs better on the *ssa*, *par*, *AIM* and binary CSPs, where as MINWGT does better on the realistic CSP problems (timetabling and nurse rostering). In addition, UTILWGT works better on *ii32* and longer term searches (larger 3-SAT and graph colouring problems), as it adds weight more slowly and so appears to maintain an effective search for longer. This result ties in with Frank's work [Frank, 1997] on causing weights to decay during the search, and it may prove useful to investigate a combination of these strategies for larger problems.

In terms of updating previous results, [Cha and Iwama, 1996] considered the performance of constraint weighting on *AIM* problems as evidence that weighting is good for especially difficult satisfiability problems. We qualify that result by showing that *AIM* problems have a recognisable structure that constraint weighting can exploit.

Randomness and Structure. In general, our findings suggest constraint weighting is better for structured problems (by structured we mean problems that exhibit noticeably different neighbour count distributions in comparison to equivalent random problems). The random problem generators used for the 3-SAT, binary CSP

and timetabling problems produce problems that are associated with smoothly increasing constraint weight graphs. These curves show a corresponding evenly increasing distribution of difficulty across constraints (from easy to hard), and so reflect the kind of problem we would expect from a purely random generator (i.e. continuous and without obvious structure). It is these problems that we would term as unstructured and for which constraint weighting has done relatively poorly. Our effort to add structure to the timetabling problem by making the domains and constraints more realistic did not produce any significant change in the performance of the various algorithms or in the shape of the constraint weight curves. However our neighbour count and small world measures did indicate we had increased the structure of the timetabling problems over the equivalent random problems. This shows that the addition of structure into a problem does not necessarily favour constraint weighting. Similarly, although the *par* problems exhibited significant structure according to our measures, constraint weighting performance for these problems was only roughly equivalent to the other methods.

Matching Problems and Algorithms. Finally, the study shows the usefulness of investigating constraint weight behaviour and problem structure when evaluating whether constraint weighting is a useful technique for a new problem domain. The presence of deviations from random neighbour count distributions, high *Ct* measures or unusual changes in the slope of a constraint weight curve, all indicate constraint weighting is at least a promising technique for a problem.

Noise Parameters. As a postscript, when considering constraint weighting as a general CSP technique it should be noted that all the weighting algorithms used in this study were parameter free (i.e. they did not have to be 'tuned' to solve a problem). In contrast (and as shown in Chapter 2), the WSAT techniques all have a noise parameter, the setting of which can significantly affect the performance of the algorithms. Similarly tabu search algorithms require the tuning of the tabu list length. Although there are various schemes for setting 3-SAT noise parameters (e.g. [McAllester *et al*. 1997; Mazure *et al*, 1997]), we found these methods were not applicable across the full range of our problem domains. Therefore considerable effort was taken to get the best performance from the WSAT and TABU algo-

rithms – effort that was not required for constraint weighting (note, there are other formulations of constraint weighting that do require parameters, for instance see [Voudouris and Tsang, 1996]).

More specifically, the WSAT noise parameters were set for this and subsequent chapters by taking a problem from each class reported and solving that problem over a range of parameter values for each algorithm. The noise value p was initially set at 50% and then varied in steps of ±10% (i.e. 50, 40, 60, 30, 80, etc). As soon as a value for p was found which had superior performance to both $(p + 10)$% and $(p – 10)$% then it was accepted as the parameter setting for that algorithm and problem class. The number of runs on a particular problem for a particular value of p was decided on a trial and error basis (depending on how clear the distinction between different p levels appeared). The tabu list length was set in a similar way but with a more variable granularity: the initial list length was set at 8 and then varied by ±1, then for list lengths > 12 we started moving in steps of 2 (14, 16, 18 etc). To allow for reproduction of our results, table 4.6 shows the various parameter settings used in the current chapter.

Problem	RNOVELTY *p*	NOVELTY *p*	BEST *p*	TABU list length
tt-struct	20	40	50	20
tt-rand	20	40	50	20
roster	40	40	50	8
bin80	40	40	50	20
bin40	40	40	50	20
g125.18	30	50	10	10
g250.15	40	10	0	20
ssa038-160	95	90	80	20
par8-2-c	90	70	40	7
par8-4-c	90	70	50	7
ii32b3	70	50	50	10
ii32c3	70	50	50	8
ii32d3	70	50	50	18
ii32e3	70	50	50	9
r100	60	60	50	5
r200	60	60	50	7
r400	60	60	50	12
AIM 100	not solved	not solved	not solved	not solved

Table 4.6. Parameter settings for WSAT and TABU algorithms

4.6 Summary

The main conclusions of the chapter are:

- Constraint weighting is best suited to problems where the weighting process is able to distinguish harder sub-groups of constraints that have a distinctly different weight profile from the remaining problem constraints.

- Constraint weighting is more likely to find these harder sub-groups of constraints within structured problems.

To recognise when constraint weighting is more appropriate we have introduced two approaches: firstly the analysis of constraint weighting behaviour using constraint weight curves and the trajectory constancy measure *Ct* and secondly the analysis of problem structure using neighbour count distributions. Finally, we found constraint weighting performance tends to degrade as problem size grows, due to a combination of larger randomised problems having less structure and a hypothesised weighting granularity effect.

Chapter 5

Improving Constraint Weighting

The empirical study in Chapter 4 shows constraint weighting to be competitive, and in some cases superior, to the latest WSAT local search heuristics. Our next question is whether constraint weighting can be improved as a general (domain independent) search technique. To address this we have developed two approaches: firstly we add a weighting capability into the WSAT and tabu search heuristics, and secondly we develop a more sophisticated weighting algorithm that places weights between constraints that are *simultaneously* violated at a local minimum. This new *arc weighting* algorithm builds on the insights of the previous chapter by recognising sub-groups of constraints that tend to become violated at the same time.

5.1 Background and Motivations

While research has been conducted into different implementations of weighting strategies [Frank 1996, 1997; Voudouris and Tsang 1996], the basic concept of constraint weighting has mainly been extended for solving satisfiability problems [Cha and Iwama 1996, Castell and Cayrol 1997]. [Frank, 1996; Frank, 1997] suggested several performance enhancing modifications to the weighting algorithm, including updating weights after each move, only changing variables that are involved in a violation, using different functions to increase weights and allowing weights to decay over the duration of the search (some of which we have tested in Chapter 4). While Frank's ideas have produced benefits in certain domains, his work can be viewed as a fine tuning of constraint weighting rather than the development of a new approach. [Voudouris and Tsang, 1996]'s work on Guided Local

Search (GLS) can also be viewed as a generalisation of constraint weighting, where constraints are 'features' and a utility function decides the weight penalties that are then translated using a paramatised cost function.

Other research has looked at new formulations of the weighting strategy. For example, [Cha and Iwama, 1996] produced significant performance improvements with their Adding New Clauses (ANC) heuristic, which instead of adding weights at a local minimum, adds a new clause for each violated clause (the new clause being the resolvent of the violated clause and one of it's neighbours). Castell and Cayrol [1997] also suggest an extended weighting algorithm called Mirror which, in addition to weighting, has a scheme for 'flipping' variable values at each local minimum. However, both ANC and Mirror are *domain dependent* techniques, ANC relying on constraints being represented as clauses of disjunct literals and Mirror requiring Boolean variables. In addition the Mirror algorithm only appears useful for a small class of problem.

In this thesis we are interested in constraint weighting as a general (domain independent) technique for solving CSPs, and in this chapter we are interested in extending or evolving the basic constraint weighting algorithm. Therefore we have not considered modifying the weighting algorithm for specific situations or the fine tuning of the weighting process. Instead we have worked to introduce something new into the basic algorithm. Firstly, we look at mixing the WSAT, TABU and weight techniques (introduced in Chapter 4) using weighted cost guidance (explained in Section 5.2). Secondly, we add another level of weighting to constraint weighting, which operates when two or more constraints become simultaneously violated in a local minimum (explained in Section 5.3). Finally we present and discuss the results of an empirical study designed to evaluate the new techniques.

5.2 Hybrid Techniques

After the promising results of Chapter 4, our next step was to see if the joining of existing techniques would produce a better overall algorithm. To do this we modified the RNOVELTY, NOVELTY, BEST and TABU algorithms introduced in Chapter 4 to use the weighted cost of a move when evaluating their respective neighbourhoods.

By adapting the MOVEWGT heuristic, we cause each of the hybrid algorithms to increment the weight of a constraint if the best move for that constraint cannot reduce the overall weighted cost. As an example, figure 5.1 shows the new GenerateLocalMoves function for the weighting version of NOVELTY, known as NOVELTYWGT (the same modifications were applied to other algorithms to produce RNOVELTYWGT, BESTWGT and TABUWGT).

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
    randomly select a violated constraint c
    BestCost ← ∞, SecCost ← ∞
    for each v_next ∈ c do
    begin
        d_curr ← current domain value of v_next
        for each d ∈ D_next | d ≠ d_curr do
        begin
            m ← {v_next, d}
            if (fw(s ⊕ m) = BestCost and LastUse(m) < LastUse(m_best))
            or fw(s ⊕ m) < BestCost then
            begin
                SecCost ← BestCost
                m_best ← m
                BestCost ← f(s ⊕ m)
            end
            elseif (fw(s ⊕ m) = SecCost and LastUse(m) < LastUse(m_sec))
        or fw(s ⊕ m) < SecCost then
            begin
                m_sec ← m
                SecCost ← fw(s ⊕ m)
            end
        end
    end
    if BestCost ≥ fw(s) then increase weight of c
    if m_best does not undo most recent change of all v_next ∈ c then m_sec ← ∅
    return m_best ∪ m_sec
end
```

Fig. 5.1. NOVELTYWGT version of GenerateLocalMoves

Figure 5.1 differs from the original NOVELTY algorithm only in respect of using a weighted cost function (*fw*) to evaluate moves and in adding weight to a constraint when no move can improve on the current overall weighted cost (i.e. **if** *BestCost* ≥ *fw(s)* **then** increase weight of *c*).

5.3 Arc Weighting

The Arc Weighting algorithm extends the concept of a weighting algorithm to include weighting the connections or arcs that exist between constraints. A simple weighting algorithm builds up weights on individual constraints each time a constraint is violated, either at a local minimum [Morris, 1993] or each time a new variable value is chosen [Frank, 1996]. In either case the weights build up a picture of how hard a constraint is to satisfy and so represent knowledge or learning about the search space [Frank, 1997]. Within this framework, weighting the arcs between violated constraints represents learning about which *combinations* of constraints are harder to satisfy. For instance, consider the example in figure 5.2:

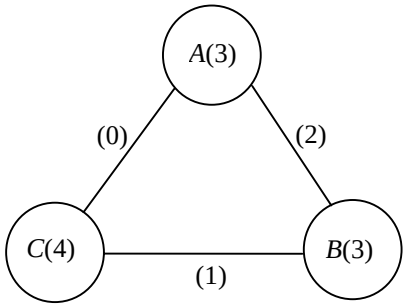


Fig. 5.2. A simple constraint weighting scenario

The nodes *A*, *B* and *C* are three constraints in a hypothetical CSP. The values associated with *A*, *B* and *C* represent the current weights on each constraint. A constraint weight equals $(1 + v)$ where v is the number of times the constraint has already been violated in a local minima and 1 is the initial weight of the constraint. (ie *A*(3) means constraint *A* has already been violated in 2 earlier local minima, plus 1 representing it's initial weight). The values on the arcs between constraints represent the number of times the two connected constraints have been *simultaneously* violated (i.e. the value 1 against arc *BC* represents that constraints *B* and *C* were once both violated in the *same* local minimum). Now consider the choice between two moves m_1 and m_2, such that m_1 violates constraints *A* and *B* and m_2 violates constraints *A* and *C*. The cost of m_1 for a simple weighting algorithm would be the sum of the weights on *A* and *B* (3 + 3 = 6) and the cost of m_2 would be the sum of the weights on *A* and *C* (3 + 4 = 7). Therefore m_1 would be preferred. However, an arc weighting algorithm would also consider that *A* and *B* have already

been violated *together* in two previous minima, so the cost of m_1 includes the arc weight *AB* (3 + 3 **+ 2** = 8). The cost of m_2 still equals 7 as the arc weight *AC* = 0. Therefore, unlike simple weighting, arc weighting would prefer m_2. In accepting m_2 the search will move to the previously unexplored area where both *A* and *C* are violated, rather than re-exploring an *AB* violation. In this way, arc weighting can produce a more diverse search that is less likely to revisit previous solutions.

Looked at more formally, arc weighting operates on a graph $G = (V, E)$ where each vertex v_i represents a constraint (or clause) and each edge e_k represents a connection between two constraints v_i and v_j. The graph is complete in order to capture all information about violated constraint groups, hence an initial set of n constraints results in a set of $n(n - 1)/2$ edges. This means a CNF satisfiability problem with 400 clauses will require 79,800 arcs! Typically an iterative repair algorithm calculates the cost of all candidate variable values before making a move. Clearly an arc weighting algorithm that checks all arcs for each variable value would be impractical with a significant number of constraints. Therefore the main challenge is to develop an efficient implementation of arc weighting without loss of arc information.

5.3.1 An Efficient Network Representation

The first step in representing the network graph is to recognise that the only relevant arcs at a particular point in the search space are those existing between currently violated constraints that have also *already* been weighted. (in the proposed algorithm, weighted constraints are those constraints that have been previously violated in a local minimum solution). Therefore the initial requirement is to build and maintain a list of currently violated, weighted constraints. This list (called *CList*) is generally short, but changes according to the type of problem and the state of the search. Next we need to represent and update the arc weights. This is done by first constructing an n x n array (called *ArcArray*) where element *i,j* represents the number of times constraints i and j have been violated together. The *CList* is then maintained in the following way: Each time a move is tested, all the newly violated and newly satisfied weighted constraints are added to a temporary list (*TList*). If the move appears promising (i.e. it satisfies at least one constraint that was previously violated) then the constraints in *TList* are merged with *CList*:

Firstly *CList* is copied (as the move may still be rejected) then each newly *satisfied* constraint is removed from *CList* and the arc weights between the satisfied constraint and each remaining *CList* constraint are calculated from *ArcArray* and subtracted from the total cost for the current move. Then each newly *violated* constraint is added to *CList* and all the arc weights between it and the existing *CList* constraints are added to the total cost. According to the new total cost, the move is either accepted or rejected. If rejected, *CList* reverts to its original state. This algorithm is shown in figure 5.3 (which shows the arc weighting cost function *fw-arc*) and figure 5.4 (which shows the arc weighting version of GenerateLocalMoves).

```
function fw-arc(CList, OldValue, NewValue)
begin
   Improve ← False, Counter ← 0, Diff ← 0, TList ← ∅
   for each constraint c_i | OldValue ∈ domain of v and v constrained by c_i do
   begin
      CChange ←  weighted cost of changing OldValue to NewValue for v
      if CChange < 0 then Improve ← True
      if c_i already weighted and CChange ≠ 0 then add c_i to TList
      Diff ← Diff + CChange
   end
   if Improve = True and TList ≠∅ then for each constraint c_i ∈ TList do
   begin
      if c_i violated with OldValue then
         if c_i satisfied with NewValue then
         begin
            delete c_i from CList
            for each constraint c_j ∈ CList do Diff ← Diff - ArcArray[i][j]
         end
      else
      begin
         for each constraint c_j ∈ CList do Diff ← Diff + ArcArray[i][j]
         insert c_i into CList
      end
   end
   return Diff
end
```

Fig. 5.3. Arc weight cost function

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
   M′ ← ∅, V_viol = ∅
   for each v_i ∈ V do if v_i in constraint violation then V_viol ← V_viol + v_i
   while M′ = ∅ and V_viol ≠ ∅ do
   begin
      select and delete v_i from V_viol
      d_curr ← current domain value of v_i
      CurrentCost ← BestCost
      for each d ∈ D_i | d ≠ d_curr and M′ = ∅ do
      begin
         m ← {v_i, d}
         CopyList ← CList
         TestCost = fw-arc(CList, d_curr, d)
         if TestCost < BestCost
         or (TestCost = BestCost and random number < p) then
         begin
            BestCost ←TestCost
            M′ ← M′ ∪ m
         end
         else CList ← CopyList
      end
   end
   if M′ = ∅ then
   begin
      MoveSideways()
      increase weights on all violated constraints and arcs
      BestCost ← BestCost + cost change due to weight increase
   end
   return M′
end
```

Fig. 5.4. Arc weight version of GenerateLocalMoves

5.3.2 Modifications to the Weighting Algorithm

As there is no ‘standard’ weighting approach, certain choices were made in the construction of the algorithm used in the study. [Frank, 1996] experimented with only testing moves for variables that are currently involved in a constraint violation. This eliminates the possibility of many ‘sideways’ moves but significantly reduces the number of values tested before each move. Tests with this approach showed a significant speed up in search times for smaller problem instances, but a tendency for the algorithm to become ‘lost’ in larger problems and fail to find a solution. A compromise approach was developed that forces a move which

changes the value of a variable not involved in a constraint violation each time a local minimum is encountered (represented by MoveSideways() in figure 5.4). For the test problems considered, this compromise performed better than either original approach.

Observation of the behaviour of the arc weighting algorithm indicated that it strongly favours solutions with only one constraint violation, and tends to cycle between these solutions (because there is zero arc weight for a single constraint violation). To remedy this behaviour an alternative weight allocation strategy was developed. Previously each constraint starts with a weight of one and is incremented by one each time it is violated at a local minimum. The new scheme distributes a fixed weight equal to the total number of constraints. If only one constraint is violated at a local minimum then it gets the full fixed weight, otherwise the weight is proportionally divided between all violated constraints. This 'proportional weighting' scheme significantly improves the performance of the arc weighting algorithm while causing the standard weighting algorithm to deteriorate.

5.4 Arc Weighting Experimental Results

5.4.1 Arc Weighting on Specialised and General Problem Domains

In a preliminary investigation we found that arc weighting generally does not perform well when embedded in an algorithm designed to exploit a specific problem structure. For instance, the fastest CNF satisfiability algorithms [e.g. McAllester *et al*. 1997] permanently store the cost of flipping each variable during the search. Each time a move is executed, the cost of flipping all connected variables is adjusted. This approach is efficient because all variables have a binary domain (true or false) and the effect of flipping a variable is easily computed by keeping a count of the number of true literals in a clause. If we add arc weighting to such an algorithm it performs badly because of the extra cost of recalculating the arc weight for all weighted variables after each move.

Similarly, a binary CSP algorithm can do a table look-up to discover whether a domain value violates a constraint (see Section 3.3.2) and consequently the cost of calculating arc-weights greatly exceeds the cost of simple constraint evaluation. Both CNF satisfiability and binary CSP algorithms exploit a binary problem struc-

ture which does not exist in realistic non-binary problems (such as nurse rostering and timetabling). Although realistic problems can be transformed into binary CSP or CNF format this can lead to large problem representations that are inefficient to solve (for example see Chapter 3 or [Cha *et al.*, 1997]).

Consequently, we have evaluated arc-weighting within a general purpose constraint satisfaction algorithm that does not assume fixed domain sizes or binary constraints (this is the same constraint engine that was used to solve the nurse rostering and timetabling problems in Chapter 4). To specifically measure the effect of arc-weighting, we developed an algorithm where the arc-weighting features can be turned off and the algorithm reverts to the MINWGT heuristic used in Chapter 4. Using these two methods (ARCWGT and MINWGT) we experimented on a larger sample of nurse rostering problems (16 problems) and transformed a selection of smaller satisfiability problems into a CSP format using the following method: a clause becomes a constraint between 3 variables, $\{x_1,x_2,x_3\}$, each with a domain of $\{0,1\}$ and corresponding coefficients $\{a_1,a_2,a_3\}$. These variables then form a constraint $a_1x_1 + a_2x_2 + a_3x_3 > b$ where a_ix_i corresponds to the i^{th} literal in the clause such that $a_i = -1$ if the literal is negative, otherwise $a_i = 1$, and $b = -$(total number of negative literals in clause). Two classes of CNF problem were used:

- randomly generated 3 SAT problems with a clause/variable ratio in the cross-over region of 4.3. These problems are prefixed with an *r* followed by the number of variables, ie *r100* represents a randomly generated, satisfiable formula with 100 variables and 430 clauses.
- single solution SAT problems with a clause/variable ratio of 2.0 created using an *AIM* generator (see [Asahiro, 1993]). These problems are prefixed *AIM* followed by the number of variables (as above).

As in Chapter 4 the nurse rostering CSPs are based on real data used to roster nurses in a public hospital. The model has a variable for each staff member, with a domain of allowable schedules. Typically there are 25-35 variables each with a domain size of up to 5000 values. Therefore the structure of the problem differs significantly from the 2 value domain of the CNF problems. In addition, approximately 400 non-binary constraints are defined between variables expressing allow-

able levels of staff for each shift, and preferred shift combinations. Although the general problem is over-constrained, optimal solutions have been found using an integer programming (IP) approach [Thornton and Sattar, 1997]. The IP solutions allow the problem to be formulated as a CSP, by defining each constraint to be satisfied when it reaches level attained in the optimum solution.

Problem	Method	RunTime (secs)					Loops	Hills	Minima
		Mean	Median	Std Dev	Max	Min			
r100	MINWGT	7.43	3.42	10.39	51	0.11	3955	3035	1398
	ARCWGT	6.04	3.72	7.90	48	0.30	2354	4218	637
r200	MINWGT	56.27	18.32	97.57	641	0.23	17171	9723	6614
	ARCWGT	20.91	9.21	27.02	121	0.93	4654	10839	1198
r400	MINWGT	342.23	292.57	202.10	1203	3.68	61534	47383	22566
	ARCWGT	79.34	59.47	69.42	602	6.31	10779	31879	2684
AIM 100	MINWGT	9.89	9.30	4.59	29	1.25	13322	5797	5101
	ARCWGT	6.41	5.62	3.49	21	1.39	7908	5796	2531
AIM 200	MINWGT	88.25	79.78	45.25	261	19.80	66072	23557	26274
	ARCWGT	56.74	48.07	38.32	268	9.40	40618	27879	13316
roster	MINWGT	144.35	29.23	250.38	1574	1.83	207	390	69
	ARCWGT	74.02	27.85	97.68	575	2.36	136	475	38

Table 5.1. Comparison of mean performance values

Problem	Time	Std Dev	Loops	Hills	Minima
r100	.81	.76	.60	1.39	.46
r200	.37	.28	.27	1.11	.18
r400	.23	.34	.18	.67	.12
AIM 100	.65	.76	.59	1.00	.50
AIM 200	.64	.85	.62	1.18	.51
roster	.51	.39	.66	1.22	.55

Table 5.2. Table 5.1 ARCWGT values as a proportion of MINWGT values

5.4.2 Arc Weighting Performance

For each category of problem, between 100 and 200 solutions were generated by each algorithm. The mean performance values for these solutions are reported in table 5.1. The Time column represents the mean execution time in seconds on a Sun Creator 3D-2000 and Std Dev is the standard deviation of the time. Loops is the mean number of iterations through the main program loop (the while loop in figure 5.4), Hills is the mean number of *improving* moves made by the algorithm and Minima is the mean number of local minima encountered.

The study uses multiple performance measures to capture precise differences between the two algorithms. While previous research has concentrated on counting the number of ‘flips’ or moves (e.g. [Cha and Iwama 1996; Frank 1996]), this

measure was found to be inadequate for comparing ARCWGT and MINWGT. As table 5.1 shows, for several problems the number of hill climbing moves made by ARCWGT exceeds MINWGT, while the ARCWGT execution time and number of iterations are actually less. This shows the number of moves is only a partial measure of the amount of 'work' done by the algorithms. The other dimension is the number of domain values tried (and hence the number of constraints tested) before a weighted cost improving move is found. This is analogous to the count of instantiations and consistency checks used in evaluating backtracking (e.g. see Haralick and Elliott [1980]). The amount of 'work' done by each algorithm is therefore better captured in counting the main program iterations (Loops in table 5.1). However, the Loops measure does not capture the extra work done by ARCWGT in maintaining *CList* (see figure 5.3). For this reason, execution times are also recorded.

5.5 Analysis of Arc Weighting

The results show that the average solution times and the average number of iterations performed by the arc weighting algorithm are significantly less than for standard weighting. This supports the earlier hypothesis that arc weighting provides additional useful information about the search space. The time results also indicate that the benefits of arc weighting outweigh the costs of maintaining the constraint list (see figures 5.5 and 5.6).

5.5.1 Distinguishing Moves

Table 5.2 re-expresses the results from table 5.1, giving the ARCWGT values as a proportion of the MINWGT values, and more clearly shows the relative differences between the algorithms. In all cases the ARCWGT results are less than the MINWGT results, except for the number of hill-climbing or improving moves. The hill climb counts are shown in more detail in figure 5.7, which plots the average number of hill climbs performed, firstly for all problems completed in less than 10,000 iterations, then for problems completed between 10,000 and 20,000 iterations, and so on. As discussed earlier, the arc weighting information should incline the search to avoid visiting previously violated groups of constraints and hence to

perform a more diverse search. The greater number of hill climbing moves combined with a reduced number of local minima for ARCWGT (see figure 5.8) indicate a more diverse search is occurring. More precisely, the hill climbing behaviour shows that, for a given number of iterations, ARCWGT is more likely to find a hill climbing move than MINWGT, because arc weighting is able to *distinguish* between moves that simple weighting would evaluate as having the same cost. Of course, the ability to distinguish between moves is only useful if the result, as in the present case, is a faster overall search.

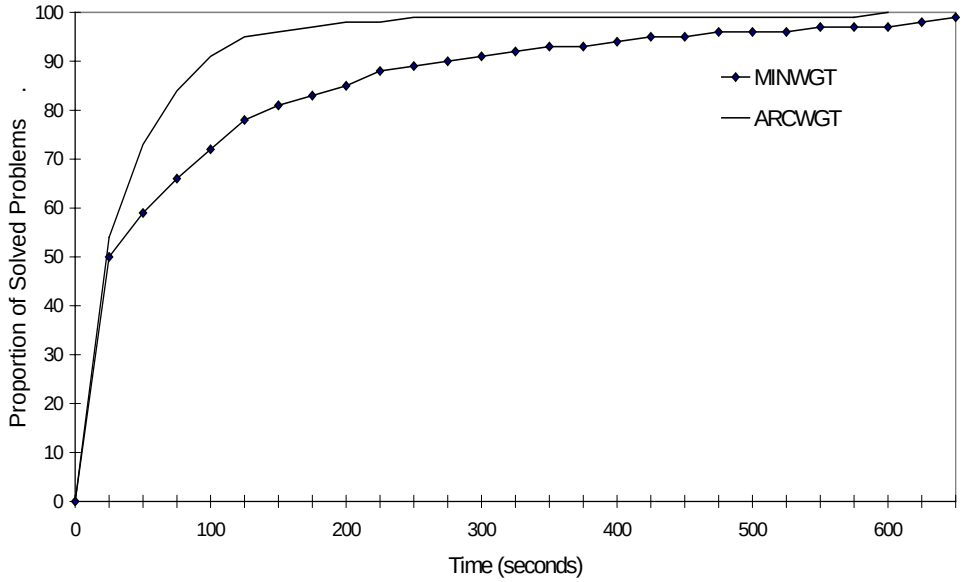


Fig. 5.5. Proportion of solved problems by time

5.5.2 Arc Weighting Costs

As would be expected, there is a greater proportional reduction in the number of program iterations than in the execution time for ARCWGT (compare figures 5.5 and 5.6). This reflects the cost to ARCWGT of using arc weights and is further analysed in table 5.3. Here the average number of iterations per second are calculated for each algorithm and problem class. The table shows the ARCWGT main loop is running at approximately 74% of the speed of MINWGT for the random CNF problems, increasing to 94% for the *AIM* problems. The probable explanation for this difference is the greater clause or constraint density for the random problems (4.3 in comparison to 2.0 for the *AIM* problems). The greater density would increase the average length of *CList* (figure 5.3) and hence add to ARCWGT's

overhead. However, a larger *CList* indicates that arc weights are also giving more guidance to the search, and so a counterbalancing improvement in search efficiency would be expected (as the results demonstrate).

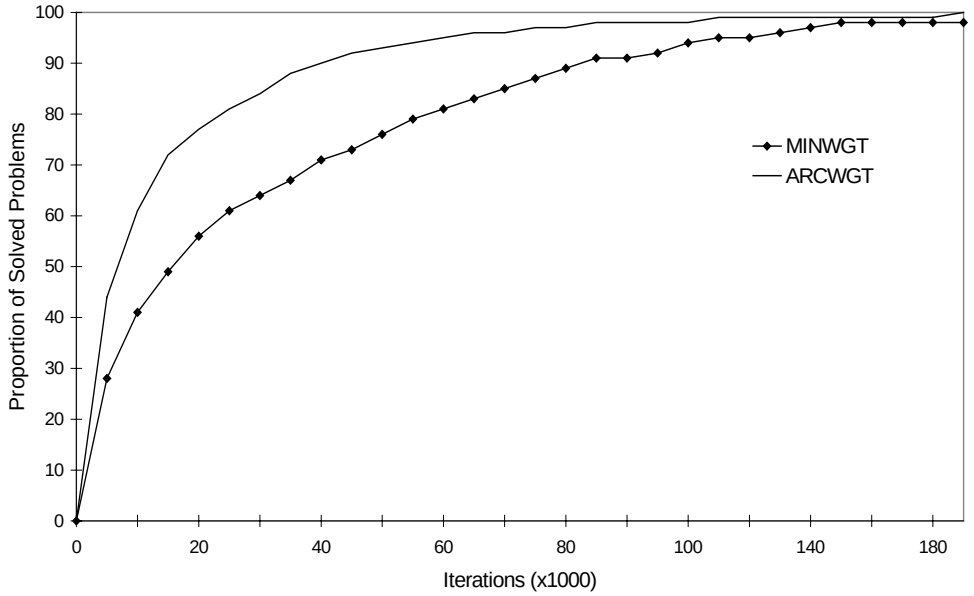


Fig. 5.6. Proportion of solved problems by iterations

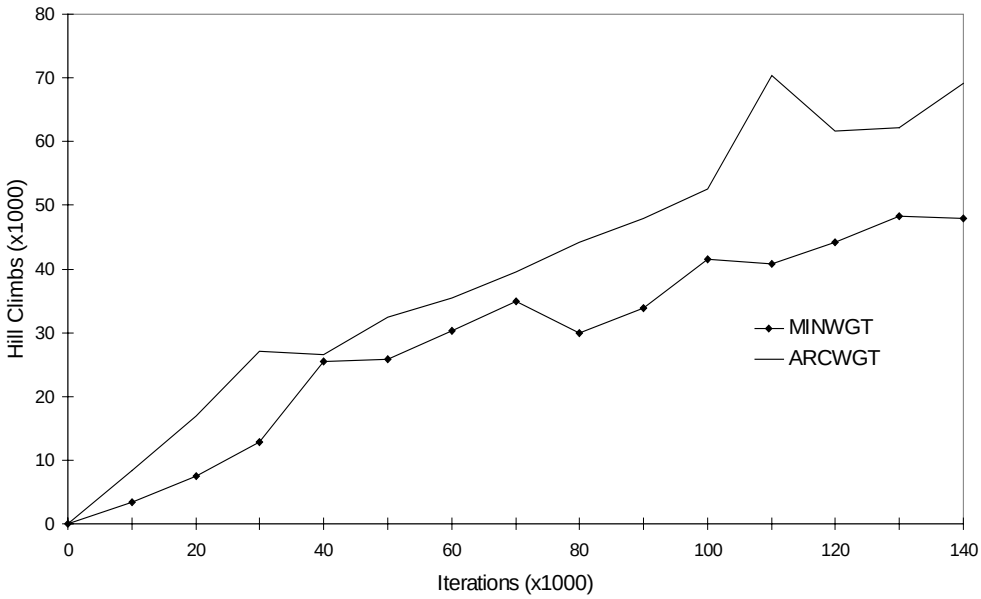


Fig. 5.7. Comparison of hill climbing moves

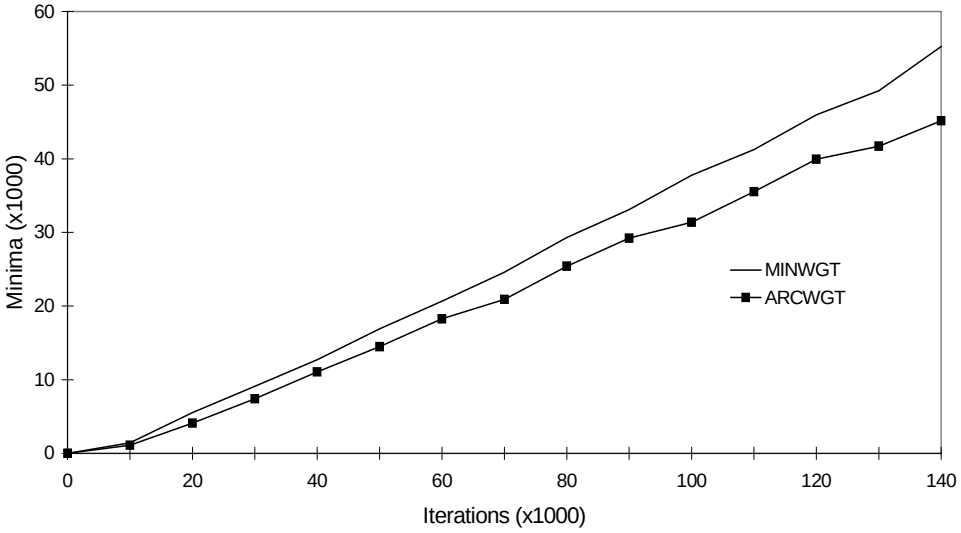


Fig. 5.8. Number of minima by iterations

Problem	Method	Loops/Time	ARCWGT/ MINWGT
r100	MINWGT	532.30	73.22%
	ARCWGT	389.74	
r200	MINWGT	305.15	72.94%
	ARCWGT	222.57	
r400	MINWGT	179.80	75.56%
	ARCWGT	135.86	
AIM 100	MINWGT	1347.02	91.59%
	ARCWGT	1233.70	
AIM 200	MINWGT	748.69	95.61%
	ARCWGT	715.86	
roster	MINWGT	1.43	92.03%
	ARCWGT	1.32	

Table 5.3. Comparison of iteration speed

5.5.3 Effects of Problem Size

The results do not support any firm conclusions as to whether ARCWGT performs proportionally better as problem size increases. While ARCWGT tends to do better with larger random CNF problems (*r200* and *r400*), there was little size effect between the single solution *AIM* problems (*AIM* 100 and *AIM* 200). However, as figures 5.5 and 5.6 indicate, the two algorithms do have similar performance when solving easier problems. This is to be expected, as in the early stages of a search, relatively few constraints are weighted. As the search continues, the number of weighted constraints grows and hence the effect of arc weighting becomes more pronounced.

5.5.4 Divergence

A further property of ARCWGT is that solution times tend to be more predictable or less divergent than for MINWGT. This is shown in the execution time standard deviations and in the graph of figure 5.9, which plots the number of solutions found at various iteration ranges. As [Cha and Iwama, 1996] point out, reduced divergence is useful when using an iterative repair technique to indicate unsatisfiability.

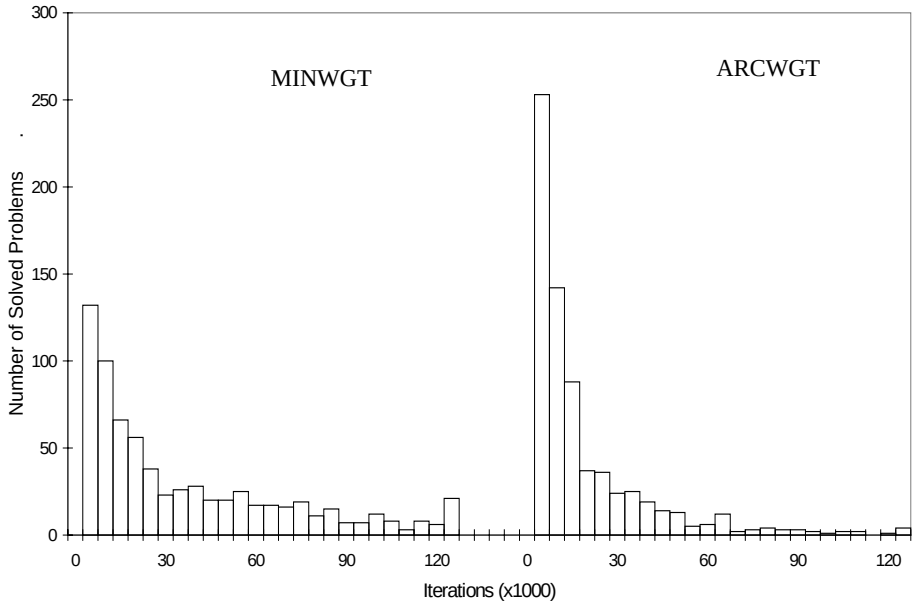


Fig. 5.9. Number of solved satisfiability problems by iterations

5.5.5 Applicability to Other Domains

As previously discussed (in Section 5.4.1), ARCWGT does not perform well in comparison to specialised weighting algorithms that exploit the binary domains of satisfiability problems and the binary constaints of binary CSPs. The current empirical study shows ARCWGT can produce performance improvements when embedded in a more general constraint solving approach using non-binary domains and constraints. The crucial questions for the arc-weighting approach are (a) whether the arc-weight information is useful to improve a search and (b) whether the gains obtained from arc-weighting can outweigh its cost. Using our results we can only conclude that arc-weighting has the better absolute performance for the nurse rostering problems. Although the arc-weighting information does improve

the search for satisfiability, the cost of maintaining the arc weights makes it uncompetitive with the specialised satisfiability algorithms used in Chapter 4.

We also looked at solving the structured timetabling problems from Chapter 4 using arc-weighting. Here we found the average execution times for ARCWGT and MINWGT to be virtually the same (100.8 seconds for ARCWGT versus 101.3 seconds for MINWGT). Again ARCWGT had a smaller Loops score (231 to 444) and a higher Hills score (3263 to 3023). More significantly ARCWGT was able to solve a greater proportion of the problems within 1 million iterations (86% versus 74%) and performed significantly better on the problems that MINWGT found hardest. In contrast MINWGT had the better performance on the easier randomly generated timetabling problems (with an average execution time of 30.2 seconds in comparison to ARCWGT's 46.5 seconds). These additional results back up similar observations from the nurse rostering problems, where ARCWGT performed better on the problems that MINWGT found relatively more difficult. This suggests that arc-weighting information is best employed in the later stages of a search after an simple weighting algorithm has failed to find an answer. Such an approach would avoid the overhead of calculating the arc-weight costs in the early part of the search (only ArcArray need be updated) but could help improve longer term searches. The issue then arises as to the exact point in a search where arc-weighting should be invoked – although this could be resolved by trial and error, further investigation may reveal that an optimum point can be calculated (based on existing weight levels in the search).

5.6 Hybrid Experimental Results

We tested our hybrid constraint weighting algorithms by directly embedding the weighting features described in Section 5.2 into the code developed by [McAllester *et al*., 1997] for solving satisfiability problems. We then re-ran the satisfiability experiments reported in Chapter 4. Results for NOVELTYWGT, RNOVELTYWGT and BESTWGT are shown in tables 5.4 (3-SAT) and 5.5 (DIMACS). Each table also reproduces the Chapter 4 unweighted results (in brackets) for each algorithm and the MOVEWGT results for each problem type. The TABUWGT results are not presented

because the weighted algorithm was unable to match or exceed the performance of TABU on any of the problem types.

Problem	Method	Flips		Time (seconds)					Success
		Mean	Cut-off	Mean	Median	Max	Min	Std Dev	
r100	NOVELTYWGT	1698 (**1331**)	250000	0.039	0.016	0.543	0.001	0.0573	100% (100)
	RNOVELTYWGT	2175 (1454)		0.052	0.018	1.204	0.001	0.0976	100% (100)
	BESTWGT	4154 (4072)		0.078	0.042	0.861	0.001	0.0956	100% (100)
	MOVEWGT	1988		0.037	0.011	2.057	0.001	0.1158	100%
r200	NOVELTYWGT	27673 (29014)	500000	0.625	0.257	10.973	0.004	0.9604	100% (99)
	RNOVELTYWGT	48833 (**25422**)		1.081	0.290	10.933	0.003	1.8638	96% (97)
	BESTWGT	74998 (46946)		1.360	0.605	8.681	0.006	1.6813	98% (99)
	MOVEWGT	50554		1.550	0.369	15.124	0.004	2.7387	86%
r400	NOVELTYWGT	183276 (108497)	1000000	4.437	2.302	24.127	0.012	5.2357	92% (94)
	RNOVELTYWGT	227159 (**85175**)		5.605	2.815	24.564	0.027	6.4395	69% (95)
	BESTWGT	304512 (147933)		5.665	3.827	18.588	0.131	5.0172	76% (93)
	MOVEWGT	175473		5.516	2.242	30.917	0.031	7.0424	62%
AIM 100	RNOVELTYWGT	6109 (-)	250000	0.085	0.045	2.112	0.004	0.1569	100% (0)
	BESTWGT	10295 (-)		0.107	0.071	1.111	0.006	0.1213	100% (0)
	NOVELTYWGT	19264 (-)		0.261	0.101	3.289	0.008	0.4407	100% (0)
	MOVEWGT	**4410**		0.085	0.041	3.066	0.002	0.2283	100%

Table 5.4. 3-SAT results for hybrid weighting algorithms

Problem	Method	Flips		Time (seconds)					Success
		Mean	Cut-off	Mean	Median	Max	Min	Std Dev	
g125.18	NOVELTYWGT	60613 (**5915**)	1000000	9.26	7.85	51.97	1.350	5.8319	100% (100)
g250.15	RNOVELTYWGT	108644 (6880)		16.36	13.15	117.32	1.590	12.2995	100% (99)
graph	BESTWGT	449189 (24566)		60.35	60.31	126.33	1.527	4.4695	52% (100)
colouring	MOVEWGT	218168		33.04	29.44	145.91	1.572	16.6267	88%
ssa	BESTWGT	5937 (29606)	500000	0.07	0.06	0.42	0.022	0.0484	100% (99)
circuit	RNOVELTYWGT	8792 (29541)		0.20	0.11	4.26	0.024	0.3070	100% (94)
fault	NOVELTYWGT	9340 (26987)		0.21	0.12	1.77	0.025	0.2364	100% (97)
diagnosis	MOVEWGT	**2885**		0.05	0.04	0.26	0.021	0.0270	100%
par8	NOVELTYWGT	**1663** (2796)	250000	0.03	0.02	0.19	0.001	0.0300	100% (100)
parity	RNOVELTYWGT	1944 (2760)		0.04	0.02	0.20	0.001	0.0347	100% (100)
function	BESTWGT	17172 (25183)		0.25	0.13	3.06	0.001	0.3792	100% (100)
learning	MOVEWGT	2542		0.05	0.03	0.87	0.001	0.0883	100%
ii32	BESTWGT	**641** (1185)	500000	0.07	0.05	0.31	0.019	0.0596	100% (100)
inductive	RNOVELTYWGT	1207 (18477)		0.17	0.13	0.60	0.028	0.1183	100% (100)
inference	NOVELTYWGT	1227 (51391)		0.18	0.12	0.83	0.029	0.1354	100% (47)
	MOVEWGT	2739		0.48	0.15	2.97	0.027	0.6633	100%

Table 5.5 DIMACS results for hybrid weighting algorithms

5.7 Analysis of Hybrid Algorithm Performance

Tables 5.4 and 5.5 show that a weighting feature in the WSAT algorithms produces varying results. Starting with the randomly generated 3-SAT problems (*r100, r200* and *r400*), weighting causes performance to deteriorate for all three hybrid techniques (NOVELTYWGT, RNOVELTYWGT and BESTWGT) in comparison to the corresponding unweighted algorithms reported in Chapter 4. Conversely, hybrid perform-

ance did improve for the *AIM* problems, where previously the unweighted algorithms were unable to find a solution. However, the hybrid WSAT algorithms could not match the performance of the pure weighting algorithm (MOVEWGT) on the *AIM* problems. As far as the 3-SAT results are concerned, the hybrid WSAT algorithms did worse on the problems the unweighted algorithms were relatively good at and better on the problems that the unweighted algorithms were relatively poor at, but were unable to produce a new best result on any of the 3-SAT problems considered.

The DIMACS results in table 5.5 produce a similar picture for the graph colouring and circuit fault diagnosis (*ssa*) problems (i.e. unweighted WSAT was good on graph colouring, now hybrid weighting causes a deterioration, and unweighted WSAT was poor on *ssa* and hybrid weighting causes an improvement, but not sufficient to outperform the pure weighting algorithm). The interesting results occur for the parity function learning problems where the hybrid NOVELTYWGT has the best overall performance of all Chapter 4 and 5 algorithms, and similarly for the inductive inference problems where the hybrid BESTWGT algorithm dominates. In both these problem domains there was little distinction between pure weighting and the best WSAT alternative reported in Chapter 4. The Chapter 5 results therefore suggest that in such circumstances (i.e. pure weighting and WSAT perform equally as well) then a hybrid technique may be applicable. However, given the limited scope of our study, this observation requires with further testing.

5.8 Summary

The main conclusions of the chapter are:

- arc weighting can lead to improved performance in solving general CSPs but is uncompetitive with techniques that exploit the binary structure of variable domains and constraints in binary CSPs and satisfiability problems.

- adding a weighting capability to the various WSAT techniques can also improve performance, most noticeably on problems where standard weighting and WSAT techniques are evenly matched.

Although the improvements we have suggested for constraint weighting can produce performance benefits, we cannot claim to have produced a better weighting algorithm in any absolute sense. The issue therefore arises as to when the various enhancements are applicable. Again the study cannot give an absolute answer to this question, but the results do suggest certain guidelines: firstly ARCWGT is applicable to more general CSP formulations that do not exhibit special problem characteristics (such as exclusively binary domains or constraints). Also ARCWGT appears to do better on harder than average problems that MINWGT has difficulty solving. As previously discussed, this suggests that arc-weighting is best employed as an 'add-on' to constraint weighting, which is invoked in the later stages of a search *within* a standard constraint weighting algorithm.

The second class of techniques considered in this chapter were the hybrid weighting algorithms. We firstly rejected the idea of adding weights to a tabu search as this consistently caused performance to decline. However, for the WSAT techniques (BESTWGT, NOVELTYWGT and RNOVELTYWGT) a weighting component did improve the performance of the original unweighted techniques reported in Chapter 4 on several problem domains. Further, for the parity function learning and inductive inference problems one of the new hybrid algorithms was able to outperform all the algorithms considered so far in the study. Interestingly, it was in these two domains that the original pure weighting and WSAT algorithms from Chapter 4 were fairly evenly matched. This led us to conclude that hybrid weighting is probably best suited to problems where neither WSAT nor pure weighting has a clear advantage.

Chapter 6

Over Constrained Problems

Real-world constraint satisfaction problems (CSPs) are often over constrained while containing a set of mandatory or *hard* constraints that *have* to be satisfied for a solution to be acceptable. Our earlier work (in Chapter 4) indicates constraint weighting performs well on problems where there is a distinction between constraint groups. Over constrained problems with hard constraints provide a ready made distinction between constraints, suggesting constraint weighting may be suitable for such problems. However, little work has been done in applying constraint weighting to over constrained problems with hard constraints. The difficulty has been finding a weighting scheme that can weight unsatisfied constraints and still maintain the distinction between the mandatory and non-mandatory constraints. This chapter presents a new weighting strategy that simulates the transformation of an over constrained problem with mandatory constraints into an equivalent problem where all constraints have equal importance, using hard constraint repetition. In addition, two dynamic constraint weighting schemes are introduced that alter the number of simulated hard constraint repetitions according to feedback received during the search.

6.1 Background and Motivations

An over constrained problem is defined as a standard CSP (i.e. as a set of variables, each with a set of domain values and a set of constraints defining the allowable combinations of domain values for the variables) with the additional proviso that *no* combination of variable instantiations can simultaneously satisfy *all* the constraints. The objective therefore becomes to satisfy *as many as possible* of the constraints [Freuder

and Wallace, 1992]. Given all constraints are of equal importance, a standard weighting algorithm can be applied to an over constrained problem with minimal modification (see section 6.2). However, most realistic over constrained problems involve constraints of *varying* levels of importance. Typically there is a set of *hard* constraints that have to be satisfied (otherwise the solution is not *acceptable*) and a set of *soft* constraints whose satisfaction is desirable but not mandatory. The simplest way to represent the relative importance of a constraint is to give it a weight. However, a weighting algorithm already applies weights to constraints during the search to escape local minima. The question then arises, how can a weighting algorithm add weights to constraints without distorting the original weights that indicate the relative importance of the constraints?

[Cha *et al*., 1997] proposed an initial answer to this question by calculating fixed hard constraint weights based on an analysis of the problem domain. The present study describes two algorithms that *dynamically* calculate the relative weights of hard and soft constraints during program execution. This means the approach is independent of specific domain knowledge and produces a more extensive search of the problem space. By analysing a set of over constrained problems, for which there are known optimal answers, the study shows the two dynamic weighting schemes perform at least as well as an ideal weight incrementing scheme that relies on foreknowledge of an optimal answer (a situation not usually found in practice).

The motivation of the chapter is to investigate the implementation of local search weighting techniques developed in the domain of CNF satisfiability to the more complex ‘real-world’ of constraint satisfaction. The chapter also introduces implementations of TABU [Glover 1989], NOVELTY and RNOVELTY [McAllester *et al*. 1997] and Guided Local Search [Voudouris and Tsang 1996] adapted to solve over constrained problems. These algorithms (plus two non-dynamic constraint weighting techniques) are compared to dynamic constraint weighting using over constrained versions of our existing nurse rostering and timetabling problems. In addition we look at a set of over constrained radio frequency allocation problems

```
procedure GenerateLocalMoves(s, TotalMoves)
begin
   M' ← ∅, BestWeightedCost ← fw(s) - δ
   for each v_i ∈ V do if v_i in constraint violation then
   begin
      d_curr ← current domain value of v_i
      for each d ∈ D_i | d ≠ d_curr do
      begin
         m ← {v_i, d}
         if f(s ⊕ m) < BestUnWeightedCost and fh(s ⊕ m) = 0 then
         begin
            BestUnWeightedCost ← f(s ⊕ m)
            BestSolution ← s ⊕ m
         end
         if fw(s ⊕ m) ≤ BestWeightedCost then
         begin
            if fw(s ⊕ m) < BestWeightedCost then
            begin
               BestWeightedCost ← fw(s ⊕ m)
               M' ← ∅
            end
            M' ← M' ∪ m
         end
      end
   end
   if M' = ∅ then IncreaseViolatedConstraintWeights()
   return M'
end
```

Fig. 6.1. GenerateLocalMoves for over constrained constraint weighting

6.2 Constraint Weighting for Over Constrained Problems

Figure 6.1 gives the pseudocode for the basic constraint weighting strategy used in the chapter. As the algorithm solves over constrained problems with hard constraints, it needs to keep track of the best solution currently found in the search (here the best solution minimises the soft constraint cost while satisfying all hard constraints). Consequently we need an additional cost function *fh(s)* that returns the number of hard constraint violations in solution *s*. This is not required for standard CSPs because a clear stopping condition exists (i.e. when *all* the constraints are satisfied). Also for over constrained problems, it is generally not known when an optimal solution is found (unless some other complete method has initially solved the problem). Instead, the search terminates when it has continued for sufficiently long without finding an

improving move. This means the terminating solution cannot be the best solution, and requires the storage of each successive best solution as it is found. Further, a constraint weighting algorithm may discover an optimum solution during the search, but fail to recognise it because the current constraint weights make another move more attractive. Therefore the algorithm must also calculate the *unweighted* cost of each move (*f(s)*) and use this measure to evaluate the best solution:

6.2.1 Weighting with Hard and Soft Constraints

As we have already shown, constraint weighting can be an effective technique for solving hard CSP problems. As yet however, there has been little work in applying constraint weighting to more realistic over constrained problems involving hard and soft constraints. A pioneering work in this area was Cha *et al*.'s paper on university timetabling [1997]. They converted a small graduate student timetabling problem into CNF format, dividing the clauses into hard and soft constraints. The hard constraint clauses were limited to being either all positive or all negative literals, reflecting the restriction that the problem of satisfying the hard constraints *must be relatively easy*. The greater importance of the hard constraints was then represented by adding a fixed weight to each hard constraint clause.

Thornton and Sattar [1997] also looked at solving a set of realistic over constrained nurse rostering problems using constraint weighting. In their approach *only* violated hard constraint weights are incremented at a local minimum. A soft constraint heuristic is then used to bias the search towards solutions that satisfy a greater number of soft constraints. However, empirical tests showed the soft constraint heuristic, although causing some improvement, was rarely able to find the (already known) optimal solutions.

Both Cha *et al*. and Thornton and Sattar's methods attempt to satisfy as many soft constraints as possible while looking for a solution that satisfies all hard constraints. Once such a solution is found, a limited search is made for the best soft constraint cost and then the algorithms are either terminated or reset. Cha *et al*. reset their constraint weights because, in further searching, the distinction between hard and soft constraints weights is lost (due to the weighting action of the algorithm) and the search is no longer able to find acceptable solutions. In Thornton and Sattar's approach the algorithm terminates because there is no mechanism that allows the soft constraint

gorithm terminates because there is no mechanism that allows the soft constraint weights to increase, so the search is unable to move out of it's local area.

Maintaining the Hard Constraint Differential. One of the contributions of this chapter is the extension of Cha *et al*.'s concept of *repeating* hard constraints [1997]. If each hard constraint is *actually* repeated in a problem (say n times) then, when a hard constraint is violated in a local minimum, all n copies of the constraint would receive a weight increment of w, causing a total increase in cost of $n \times w$. This can be simulated, as with Cha *et al*., by giving each hard constraint an initial weight of n. The new step is to increment each *hard* constraint violated at a local minimum with a weight of $n \times w$ instead of w (soft constraint violations are still incremented by w). Such a system behaves identically to a system where all constraints have equal weight, with each hard constraint repeated n times. Previous studies have already demonstrated that simple constraint weighting is an effective search strategy. Therefore we should expect the new hard constraint weighting strategy to be equally effective.

In order to adequately explore the search space, a constraint weighting algorithm must be able to move from one area to another where all hard constraints are satisfied, *via intermediate solutions where some hard constraints are violated*. Unlike the previously discussed algorithms, the new hard constraint weighting strategy is able to do this *systematically* rather than accidentally:

Example. Consider the situation in figure 6.2: *A, B, c* and *d* represent four constraints in an unspecified over constrained problem, where *A* and *B* are hard constraints, *c* and *d* are soft constraints, and w_A, w_B, w_c and w_d represent the constraint weights of *A, B, c* and *d* respectively. Let the number of hard constraint repetitions $n = 3$ and the weight increment $w = 1$. Hence, the soft constraints are given initial weights $w_c = w_d = w = 1$, and the hard constraints are given initial weights $w_A = w_B = n \times w = 3$. Figure 3(a), represents the first local minimum found in the search, where all hard constraints are satisfied and both soft constraints are violated. As yet no weights have been added by the search so the cost of the solution = $w_c + w_d = 2$. A constraint weighting algorithm will now add weight w to *c* and *d*, making $w_c = 2$ and $w_d = 2$, and a new solution cost = 4. If we assume there is no move available that does not violate both hard con-

straints, then we are still at a local minimum (as $w_A + w_B > w_c + w_d$) and the soft constraints will be incremented twice more until $w_c = w_d = 4$. In this case the cost of violating both hard constraints (6) is less than the cost of violating both soft constraints (8), so the move which violates both hard constraints will be accepted (shown in figure 6.2(b)). Assuming this solution is another local minimum, the weights of *a* and *b* are now incremented. In Cha *et al.*'s scheme, w_A and w_B will be incremented by w to 4 (figure 6.2(c)), whereas in the new constraint weighting scheme w_A and w_B will each be incremented by $n \times w$ to 6 (figure 6.2(d)). Here the crucial difference between the two approaches is evident. In Cha *et al.*'s solution all constraints now have the same weight and *there is no way to further distinguish between the hard and soft constraints*. This means the search has no guidance towards solutions which satisfy the hard constraints. In the new constraint weighting strategy, the soft constraints have been allowed to overpower the hard constraints, but as soon as a hard constraint is violated the dominance of the hard constraints is reasserted and the search will now concentrate on finding *another* solution where all hard constraints are satisfied.

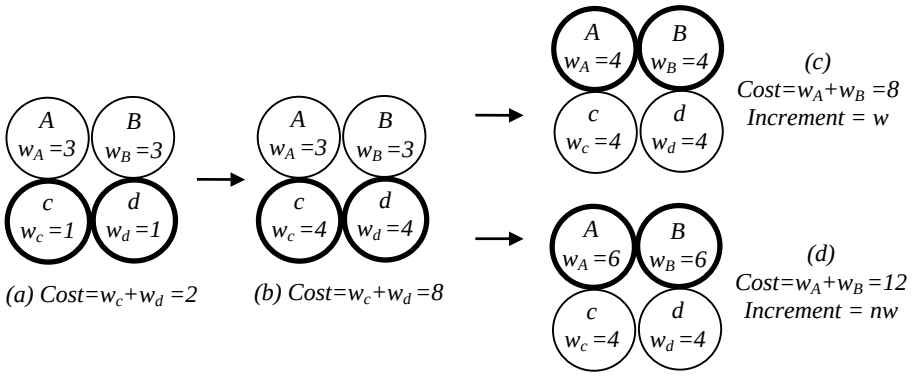


Fig. 6.2. Weighting hard and soft constraints

Deciding the Initial Hard Constraint Weights. [Cha *et al.*, 1997] recognised the crucial question for their research was to find the best number of repetitions of the hard constraint clauses. In the extreme case, the weight on each hard constraint can be set to equal the total initial cost of violating all soft constraints plus one (as in the previous example). However, such a scheme when applied to their timetabling problem places very large initial weights on the hard clauses. In practice they found the hard constraint clauses are quickly satisfied with such weights, but high levels of soft constraint violation remain. At the other extreme, giving insufficient weight to the hard

constraints results in a search that is unlikely to find *any* solution where all hard constraints are satisfied (although soft constraint satisfaction would be very high).

The issue of the number of repetitions is equally important to the new constraint weighting scheme. The greater the difference between the initial hard and soft constraint weights the slower the search will be, as it will take longer to build up weights on the soft constraints. However, setting the initial hard and soft constraint weights too close together will cause the search to excessively explore areas of hard constraint violation where (by definition) no acceptable solution can exist. Worse still, the search may approach an optimum solution but fail to converge on it because of the over-valuing of the soft constraints. The question therefore arises, how much weight is too much and how much is too little? Cha *et al.*'s answer was to look at their particular problem and calculate the average number of soft constraint violations that would be caused by satisfying a currently violated hard constraint (they assume that most constraints are already satisfied). They then use this value to set the initial hard constraint weights. Clearly there are problems with this approach. Firstly, the number of soft violations caused by the satisfaction of a hard constraint will vary within the search space and secondly, the method requires a detailed analysis of the search space.

6.2.2 Dynamic Constraint Weighting

A useful property for a hard and soft constraint weighting algorithm would be the ability to learn the correct ratio of hard to soft constraint weights *during the search itself*. Consequently, the second contribution of the chapter is the development and empirical evaluation of two such dynamic constraint weighting strategies.

Downward Weight Adjustment (DWA). The first strategy, Downward Weight Adjustment, involves starting the search with the number of repetitions, n, set to the total number of soft constraints + 1 (i.e. the maximum value). Then, as soon as a solution is found where all hard constraints are satisfied (i.e. an *acceptable* solution), the value of n is adjusted downwards to be one more than the number of soft constraints currently violated (s_{cur} + 1). Each time a new acceptable solution is found such that $s_{cur} < n$, then n is set to s_{cur} + 1 (the new best level of soft constraint violation), i.e. the number

of hard constraint repetitions is *dynamically* adjusted according to the best solution found so far in the search.

This approach is based on the insight that number of hard constraint repetitions, n, should not be set to less than the *optimum* number of soft constraint violations, s_{opt}. If n is less than s_{opt} then the search will tend to prefer a solution where a hard constraint is violated over an optimal solution. If n is close to but greater than s_{opt} then the search may prefer a single hard constraint violation over many *non*-optimal acceptable solutions, but will still prefer an optimal solution. However the value of s_{opt} is generally unknown (unless a complete method has already solved the problem). Therefore, Downward Adjustment Weighting keeps making a closer and closer estimate of s_{opt} by resetting the value of n each time a new *unweighted* cost reducing solution is found. However, the definition of unweighted cost has become more complex due to introduction of constraint repetition. Now the unweighted cost equals the *number* of violated constraints *including* repetitions and the weighted cost equals the *sum of the weights* of all violated constraints *including* repetitions. Put more formally, consider an over constrained problem with a set of hard constraints $H = \{h_1, h_2, h_3, ... h_k\}$ and a set of soft constraints $S = \{s_1, s_2, s_3, ... s_j\}$. Each hard constraint has a weight wh_i, $i = 1... k$, and each soft constraint has a weight ws_i, $i = 1 ... j$, where the weight i equals the number of times constraint i has been violated in a local minimum. Letting n be the number of hard constraint repetitions, CH be a vector with elements ch_i, where $i = 1... k$, such that element $ch_i = 0$ if h_i is satisfied and $ch_i = 1$ otherwise, and CS be a vector with elements cs_i, where $i = 1 ... j$, such that element $cs_i = 0$ if s_i is satisfied and $cs_i = 1$ otherwise, then we have the following definitions:

$$\text{WeightedCost} = n \sum_{i=1}^{k} wh_i\, ch_i + \sum_{i=1}^{j} ws_i\, cs_i \qquad (1)$$

$$\text{UnweightedCost} = n \sum_{i=1}^{k} ch_i + \sum_{i=1}^{j} cs_i \qquad (2)$$

The analysis so far assumes a weight increment of one and that constraints have only two states: satisfied or violated. However, the approach can be easily extended to include different additive or multiplicative weight increments and varying levels of constraint violation.

Flexible Weight Adjustment (FWA). The second dynamic constraint weighting strategy involves adjusting the value of *n* according to the current state of the search. We start with the smallest differential that distinguishes hard and soft constraints (i.e. $n = 2$) and then proceed to increase the value of *n* by 1 each time a non-acceptable local minimum is encountered. *n* is therefore increased to a level sufficient to cause all hard constraints to be satisfied. Each *acceptable* local minimum encountered, causes *n* to be reduced by 1, making it easier for hard constraints to be violated and so encouraging the search to diversify out of the current local area. In effect, in non-acceptable areas the search becomes increasingly attracted to acceptable areas and in acceptable areas the attraction moves to the non-acceptable. Using the earlier definitions of n, h_i, s_i, wh_i and ws_i, figure 6.3 gives the pseudocode necessary to implement FWA (Note IncreaseViolatedConstraintWeights() is called from the main constraint weighting algorithm in figure 6.1).

```
procedure IncreaseViolatedConstraintWeights()
begin
   TotalHardViolations ← 0
   for each violated hard constraint h_i do
   begin
      wh_i ← wh_i + 1
      TotalHardViolations ← TotalHardViolations + 1
   end
   for each violated soft constraint s_i do ws_i ← ws_i + 1
   if TotalHardViolations > 0 then n ← n + 1
   else if n > MinRepetitions then n ← n − 1
end
```

Fig. 6.3. The Flexible Weight Adjustment algorithm

6.3 Experiments

6.3.1 Control Algorithms

The two dynamic weighting strategies were compared to two forms of fixed weighting called MaxIncrement (MAX) and MinIncrement (MIN). MaxIncrement sets the weights of all hard constraints to the total number of soft constraints plus one, and increments all hard constraints by this amount in a local minimum. This is the largest realistic setting for the constraint increment and favours solutions where all hard constraints are satisfied at the expense of satisfying the soft constraints. MinIncrement

sets the weights of all hard constraints to the number of soft constraints left unsatisfied in an *optimal* solution (plus one) and again increments by this value. The optimum level of constraint violation is the smallest realistic setting for an increment, otherwise the search is likely to ignore an optimum solution (see section 6.2). An implementation of [Cha *et al*., 1997]'s reset algorithm was also tried on our test problems, but in most cases the algorithm was unable to find an acceptable solution. Cha *et al*.'s approach assumes the initial problem of finding an acceptable solution is relatively easy. In our test problems this was not the case.

6.3.2 Comparison Algorithms

We further developed versions of the BEST, TABU, NOVELTY, and RNOVELTY algorithms (introduced in Chapter 4) for over constrained problems. For each of these techniques (as in MinIncrement) the cost of violating a hard constraint is set to be n + 1 times as great as violating a soft constraint, where n is the least number of soft constraint violations found by any of the weighting techniques. This extends earlier work by [Jiang *et al*., 1995] on using WSAT to solve weighted MAX-SAT problems. We also tested the dynamic weighting scheme used by [Schaerf, 1996] for tabu search. In this method, the weight on a class of constraints is incremented if any of the constraints are violated after a fixed cycle of moves, otherwise the weights are decremented. Applying this scheme to the hard constraints in our test problems produced poor results because (in most cases) the problem of satisfying the hard constraints was difficult in itself. Consequently, continually increasing weights tended to build up on the hard constraints, producing poor sensitivity to changes in soft constraint costs. For this reason further investigation of the scheme was rejected. Adaptations of DWA and FWA for TABU, NOVELTY and RNOVELTY were also tried but did not perform as well as the fixed weight method. Consequently only the fixed weight versions of TABU, NOVELTY and RNOVELTY are reported. Additionally, the performance of BEST was significantly inferior on all problem domains, so for brevity we have removed BEST from the results and discussion.

Finally we developed a version of [Voudouris and Tsang, 1996]'s Guided Local Search (GLS). This extends the UTIL algorithm of Chapter 4 to include consideration of the differential cost of hard and soft constraints, and introduces a new cost function

that uses both the unweighted and penalty cost of a solution. Previously UTIL penalised (or weighted) constraints in a local minimum with the greatest utility, as measured by:

$$utility_i(s^*) = I_i(s^*) \times (c_i / (1 + p_i))$$

where s^* is the current solution, i identifies a feature, c_i is the cost of feature i, p_i is the penalty (or weight) currently applied to feature i and $I_i(s^*)$ is a function that returns one if feature i is exhibited in solution s^* (zero otherwise). For CSPs we assumed all constraints to be features with a cost of 1. Now with hard and soft constraints we can define a greater cost (c_i) for the hard constraints and so further bias the search to satisfy these constraints (as these constraints will also attract greater penalties). Further, we use the GLS cost function of the form:

$$cost(s^*) = g(s^*) + \lambda \Sigma p_i I_i(s^*)$$

where p_i, I_i and s^* are defined as before, $g(s^*)$ is the unpenalised cost of s^* and λ is a parameter defined within the GLS algorithm. To maintain consistency between algorithms we embedded GLS within our standard local search neighbourhood function which randomly selects variables involved in constraint violations (this differs from [Voudouris and Tsang, 1996]'s Fast Local Search selection method). Also we increment penalties on constraints by one in a local minimum whereas GLS uses a more sophisticated scheme based on the maximum or minimum cost difference in the local neighbourhood of moves.

6.3.3 Test Problems

The algorithms were tested on the set of 16 real-world nurse rostering problems and 10 randomly generated timetabling problems introduced in Chapter 4 and also on a selection of the Radio Link Frequency Assignment Problems (RLFAP) used by [Voudouris and Tsang, 1996] to evaluate GLS.

The nurse rostering problems involve allocating a set of pre-generated legal schedules to each nurse in a roster, such that all hard constraints involving levels of staff for each shift are satisfied. Soft constraints are introduced by attaching a score to each schedule indicating how *unattractive* a schedule is for a nurse (where the best schedule gets a zero score). Therefore each nurse is associated with one soft constraint

which is satisfied if the nurse is allocated a zero cost schedule. Further details of the problems are described elsewhere [Thornton, 1995]. One attractive feature of the domain is that, although the problems are difficult for a local search algorithm to solve, we have optimal answers for each problem obtained from an integer programming application [Thornton, 1995].

The over constrained timetabling problems were generated by adding additional soft constraints for each staff member defining which time slots the staff member would *prefer* to teach (existing hard constraints already define which time slots the staff member is unavailable to teach). These soft constraints were added to the randomly generated timetabling problems (*tt_rand*) already reported in Chapter 4.

Finally we used a selection of the over constrained RLFAPs reported by [Voudouris and Tsang, 1996] in their evaluation of GLS. The chosen problems (1, 2, 3 and 11) involve assigning frequencies to radio links subject to a set of hard binary constraints defining the minimum or exact difference in frequencies between radio link pairs. Additionally the problems have an optimisation criteria that a solution should use the least possible number of frequencies. If we think of this criteria as a constraint, we want the total number of frequencies assigned in a solution to equal *m*, where *m* is the (unknown) optimum frequency use. This constraint involves a *dynamic* assignment cost and so cannot be modelled as a simple relationship between variables or by assigning a fixed cost to a domain value (as in the nurse rostering problem). Instead we have to consider the number of times a particular frequency (domain value) is assigned for each move. [Voudouris and Tsang, 1996] take the cost of using domain value v to be $NVar - NVar_v$ where $NVar$ is the total number of variables and $NVar_v$ is the number of variables using value v. This cost is then included in the overall move evaluation cost. Additionally, penalties are associated with each v and may be increased at a local minimum according to the utility function defined earlier. The problem with Voudouris and Tsang's approach for our neighbourhood selection heuristic is that there is no criteria to decide whether a variable is in violation of an assignment cost (in effect the constraint is to make all assignment costs equal zero hence all variables are in violation). For this reason we added the definition that a variable is only in violation of an assignment cost constraint if it is currently assigned the *least used frequency*. Similarly, only the least used frequency is weighted at a local minimum.

6.3.4 Results

All problems were solved on a Sun Creator 3D-2000. For the nurse rostering problems, runs were either terminated on finding an optimum solution, or after 1 million domain value changes had been tested. The timetable problems, having no known optimal solution, were also terminated after 1 million domain value tests. The RLFAPs were solved using an adapted *binary* CSP algorithm and were terminated on finding an optimal solution (as reported by [Voudouris and Tsang, 1996]) or after 200,000 variable tests (approximately 3 million domain value tests).

Tables 6.1 to 6.3 show the average scores, times, values tested and proportion of problems solved for each algorithm. The nurse rostering results show the average of 160 runs (10×16 problems), the timetabling results show the average of 100 runs (10×10 problems) and the RLFAP results show the average of 400 runs (100×4 problems) for each algorithm. For each problem instance the mean, median and standard deviations were calculated over all runs for a particular algorithm. The results then show the *averaged* statistics over all problems in a class for each algorithm (Note, the proportion of optimal solutions is not reported for the timetabling problems because the optimum solution score is unknown). The result tables also report statistics on all runs and for successful runs. By successful we mean those runs that found a solution satisfying all hard constraints. Those runs which failed to find such a solution were given a cost or score of 100. Otherwise scores refer to the best level of soft constraint violation found within a hard constraint satisfying solution.

Generally, the results for over constrained problems with unknown optimum solutions are best interpreted using anytime curves [Freuder and Wallace, 1992]. These curves plot the cost of the best solution found in the search against execution time, and represent the quality of solution that would be found if an algorithm were terminated at a particular point. Anytime performance is significant for problems where there is insufficient time to find an optimal solution, or the optimum is unknown, and so are relevant to over constrained problems. Consequently we have also generated anytime curves for each problem domain, firstly comparing the performance of our weighting and control algorithms (FWA, DWA, MAX and MIN) and then comparing the best of the weighting strategies with our other techniques (NOVELTY, RNOVELTY, TABU and GLS). These curves are shown in figures 6.4 to 6.9. In each

graph, the y-axis represents the averaged sum of all soft constraint costs of the best solutions found at a given time for each run of an algorithm (as before, a solution that violates a hard constraint is given a fixed cost of 100).

Rostering	FWA	DWA	MAX	MIN	GLS	NOV	RNOV	TABU
% solved	95.63	100.00	99.38	98.13	96.88	67.50	66.88	56.25
% optimal	70.00	70.63	44.38	76.25	30.63	21.25	21.88	9.38
All Runs								
best score	21.69	21.81	22.31	21.94	23.38	36.50	35.50	45.19
median score	23.12	22.53	23.53	23.54	25.91	42.84	46.53	54.66
mean score	25.17	22.78	24.40	24.19	27.83	47.02	47.29	54.67
std deviation	4.13	1.02	2.77	2.06	6.15	12.44	9.74	8.44
Successful Runs Score:								
best	21.69	21.81	22.31	21.94	23.38	21.85	20.62	20.27
median	22.31	22.53	23.50	23.28	25.84	24.81	25.46	23.95
mean	22.79	22.78	24.09	23.26	26.16	25.38	25.36	24.13
std deviation	2.08	1.49	2.24	1.74	2.68	3.22	3.62	4.17
Time (secs):								
median	111.28	134.81	177.03	111.78	82.75	167.73	148.96	97.41
mean	126.05	142.49	185.89	125.61	107.24	168.72	166.47	101.90
std deviation	301.71	292.95	288.99	280.93	260.27	289.22	282.25	174.25
Values tested:								
median	260545	323393	425935	269591	203049	1697130	1557874	1017677
mean	294565	342642	449777	303266	266163	1708292	1725057	1096642
std deviation	243287	232773	235305	219788	219145	1060563	1049648	740826

Table 6.1. Averaged results for 16 nurse rostering problems

6.4 Analysis

6.4.1 Nurse Rostering

Starting with the nurse rostering problems, table 6.1 shows all the weighting algorithms (FWA, DWA, MAX, MIN and GLS) performing well in terms of finding solutions (with success rates ranging from 95.63% to 100%) where as the non-weighting techniques (TABU, NOVELTY and RNOVELTY) have trouble finding any hard constraint satisfying solutions (with success rates ranging from 67.5% to 56.25%). If we further consider the % optimal measure we see only FWA, DWA and MIN are consistently able to find optimal answers (with success rates ranging from to 70% to 76.25%). The score (solution cost) statistics do little to separate the better algorithms (FWA, DWA and MIN), except DWA does have a slightly lower mean and standard deviation. Also, although TABU has the lowest average best score for successful runs (20.27), this does not indicate superior performance, as TABU is only able to find solutions slightly more than half the time (56.25%) and had the worst rate for finding

optimal solutions (9.38%). Similarly, a reading of the time and values tested statistics suggests that TABU is doing better because it is faster. However, these measures indicate the point at which an algorithm *stops finding improving solutions*, showing only that TABU is more effective in the earlier stages of the search. The best picture of the relationship between time and score is given by the anytime curves in figures 6.4 and 6.5. Here (in figure 6.4) we see the initial weighting techniques (FWA, DWA, MAX and MIN) having fairly similar performance, but with DWA doing slightly better in the longer term (as suggested by the score statistics). Figure 6.5 plots DWA against the other techniques (GLS, TABU, NOVELTY and RNOVELTY) and shows the weighting strategies (DWA and GLS) are doing significantly better than the other techniques, with DWA again dominating. Interestingly, the curves show the non-weighting techniques (TABU, NOVELTY and RNOVELTY) are clustered together with similar shaped curves, indicating a clear distinction between the weighting and non-weighting techniques for this domain.

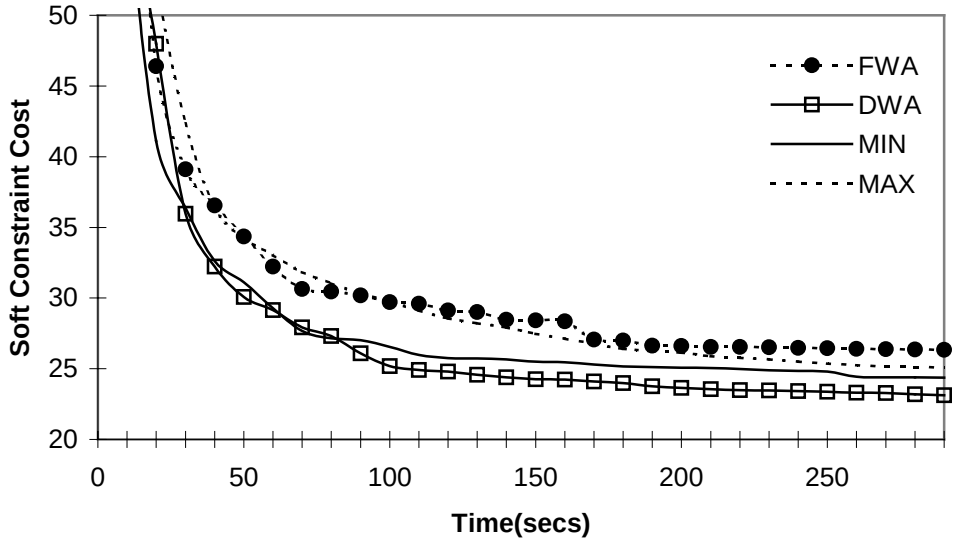


Figure 6.4. Nurse rostering anytime curves for weighting algorithms

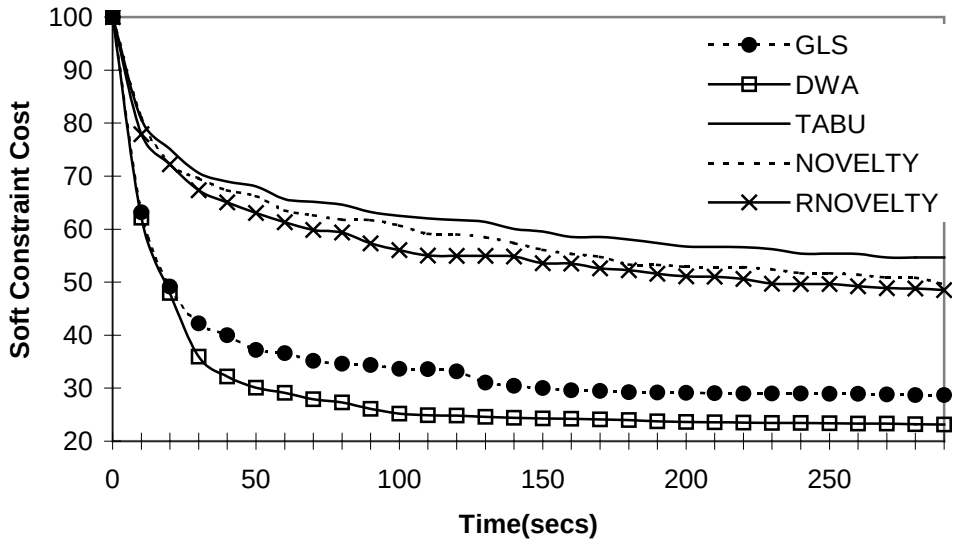


Figure 6.5. Nurse rostering anytime curves for comparative algorithms

6.4.2 Timetabling

The timetabling statistics in table 6.2 again show the weighting strategies are doing better in terms of the percentage of solutions found, although GLS is doing less well than the other weighting strategies (66% versus 85-96%). More generally, NOVELTY and RNOVELTY are doing noticeably worse than any of the other techniques with a 21-30% success rate. The score statistics for all runs start to separate FWA and MIN as the better techniques, with TABU appearing the next most promising (again the low scores for NOVELTY and RNOVELTY on successful runs are explained by the low success rate for these algorithms). The time and values tested statistics do little to distinguish between techniques except to indicate that NOVELTY and RNOVELTY tend to stop finding improving solutions earlier in the search. Again the anytime curves in figures 6.6 and 6.7 give the clearest picture. Figure 6.6 distinguishes FWA as the better weighting technique, and (in comparison to the nurse rostering curves of figure 6.4) we see a greater separation between weighting strategies. In looking at the broader category of techniques in figure 6.7, FWA again strongly dominates the other methods. However, in comparison to the nurse rostering curves (figure 6.5) GLS performs quite poorly and TABU proves to be the better of the non-weighting techniques.

Timetabling	FWA	DWA	MAX	MIN	GLS	NOV	RNOV	TABU
% solved	92	91	96	85	66	21	30	59
All Runs								
best score	35.70	58.90	71.30	41.80	61.80	65.90	52.80	49.10
median score	48.85	76.25	85.50	59.05	81.20	89.20	86.95	69.30
mean score	50.56	77.18	85.46	60.43	81.71	87.63	82.58	69.41
std deviation	11.36	13.56	8.46	14.87	14.12	13.25	15.95	18.63
Successful Runs Score:								
best	35.70	58.90	71.30	41.80	57.56	31.80	32.57	43.44
median	47.60	76.65	85.60	57.00	73.94	37.50	39.29	49.56
mean	47.64	77.23	85.53	57.49	73.47	37.83	39.20	49.88
std deviation	7.96	14.20	8.56	13.19	11.34	5.72	7.77	5.05
Time (secs):								
median	237.45	216.10	225.65	229.05	216.72	184.64	173.03	222.38
mean	232.55	211.23	212.79	222.56	211.50	185.88	168.77	207.84
std deviation	79.49	98.78	140.99	106.26	105.22	83.42	104.77	131.00
Values tested:								
median	922419	828301	840790	854674	812718	715688	648981	795806
mean	889094	815197	786014	825519	789885	721987	634235	739870
std deviation	105495	139212	176849	139223	157820	141324	148944	200714

Table 6.2. Averaged results for 10 random timetabling problems

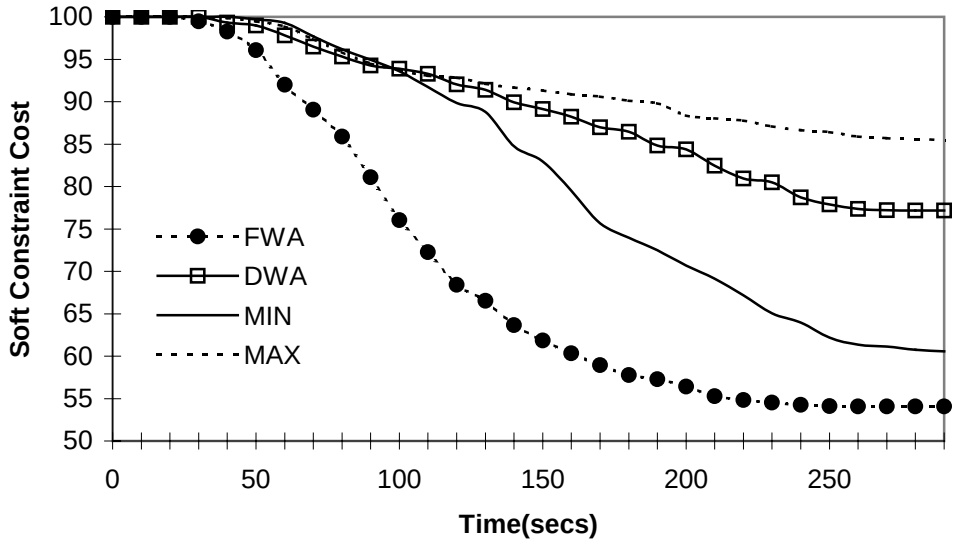


Figure 6.6. Timetabling anytime curves for weighting algorithms

6.4.3 RLFAPs

Unlike the other problem domains, the % solved and % optimal values for RLFAP in table 6.3 do not strongly distinguish between techniques. DWA has the highest success rate (97%) in a range of 84.5 to 97 and GLS finds more optimal solutions (40.75%) in a range of 32.5 to 40.75. Additionally the score statistics show a roughly equivalent performance between algorithms (for instance, the mean score for all runs

ranges from 25.96 to 32.24). Similar time and values tested statistics are also observed, although TABU does tend to continue improving for longer than the other techniques. The anytime curves in figure 6.8 show FWA to have the better initial performance although MIN and DWA do approach and meet FWA in the later stages of the searches. Figure 6.9 shows a greater separation with FWA performing better than GLS and the other non-weighting techniques (TABU, NOVELTY and RNOVELTY).

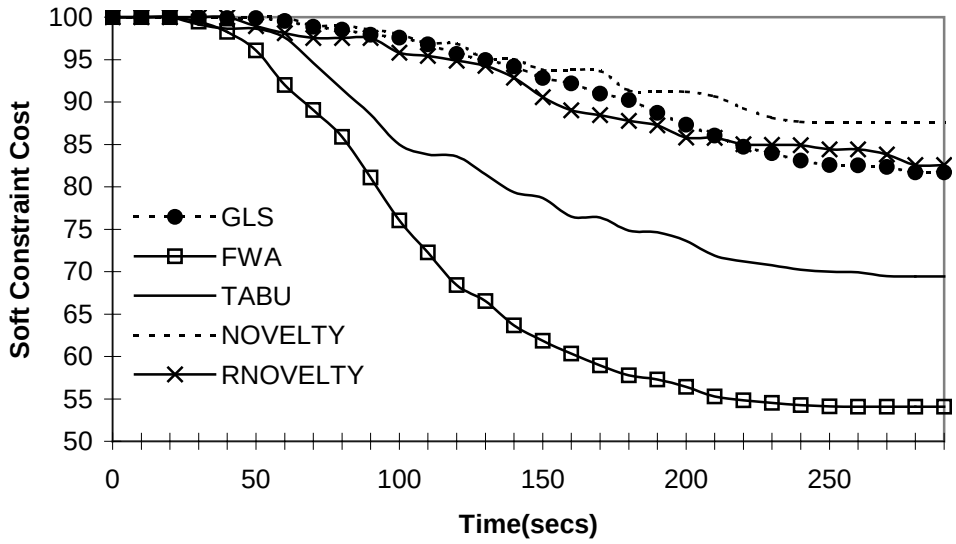


Figure 6.7. Timetabling anytime curves for comparative algorithms

RLFAPs	FWA	DWA	MAX	MIN	GLS	NOV	RNOV	TABU
% solved	95.75	97.00	95.50	95.5	88.00	92.50	92.50	84.50
% optimal	36.25	37.00	33.25	37.5	40.75	32.50	34.50	34.00
All Runs								
best score	20.00	20.00	20.50	19.5	20.50	19.00	19.00	18.50
median score	23.00	24.00	24.50	23	23.50	22.25	21.75	38.00
mean score	26.24	25.96	27.74	26.42	30.80	27.08	27.23	32.42
std deviation	8.12	7.51	8.17	8.28	9.79	9.29	9.60	10.76
Successful Runs Score:								
best	20.00	20.00	20.50	19.5	20.50	19.00	19.00	18.50
median	23.00	24.00	24.50	23	23.50	22.25	21.75	21.00
mean	23.76	24.21	25.12	23.79	23.99	22.31	22.27	22.31
std deviation	4.20	4.47	4.10	4.32	4.01	3.14	3.29	5.19
Time (secs):								
median	4.93	6.70	7.18	4.75	4.33	7.95	8.12	13.10
mean	8.36	10.56	11.31	9.22	7.93	14.27	16.63	19.61
std deviation	123.97	123.60	129.29	125.47	101.31	193.61	224.37	221.40
Values tested:								
median	825071	1012192	1075649	716451	672930	958040	958170	1484668
mean	1241642	1413781	1527254	1182376	1126668	1703293	1940451	2126074
std deviation	1673644	1460627	1537160	1463000	1542761	2502318	2854597	2831966

Table 6.3. Averaged results for 4 RLFAPs (1,2,3 and 11)

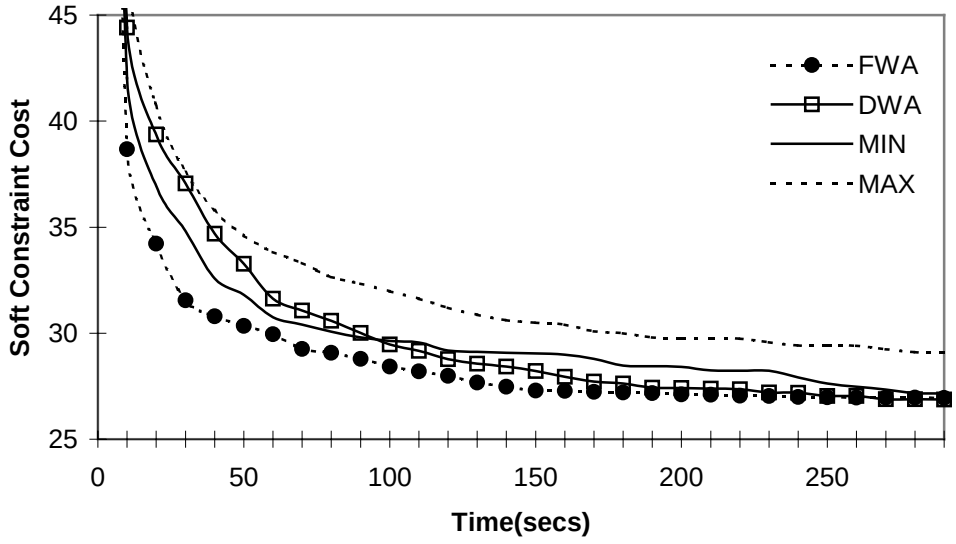


Figure 6.8. RLFAP anytime curves for weighting algorithms

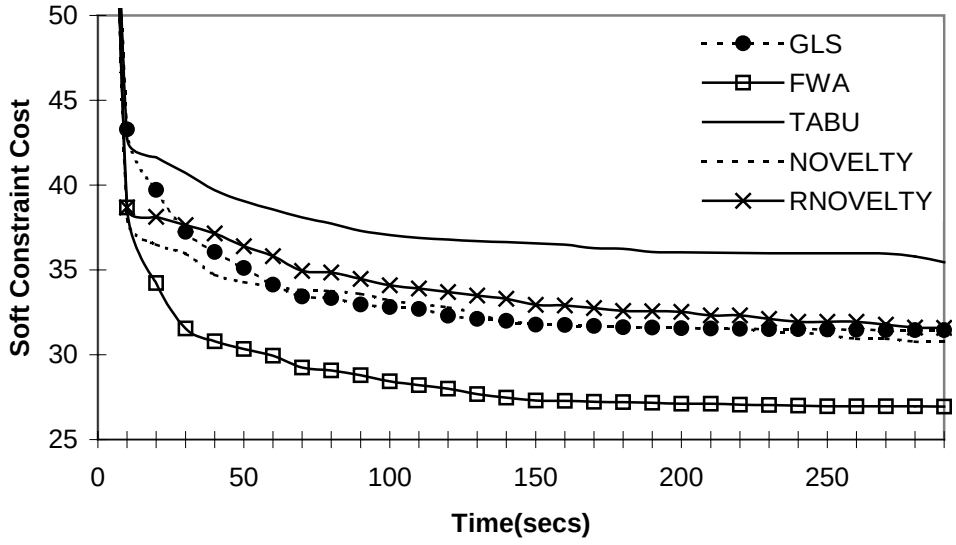


Figure 6.9. RLFAP anytime curves for comparative algorithms

6.4.4 Overall Comparison

The results, summarised by the anytime curves, show that the two dynamic weighting strategies (FWA and DWA) dominate in all three problem domains. FWA is better on timetabling and RLFAP and DWA is better on the nurse rostering problems. These results should be compared to the results in Chapter 4 (table 4.3) which are based on the *same* nurse rostering and timetabling problems formulated as CSPs. In the Chapter

4 nurse rostering CSP formulation the schedule cost is modelled as a *hard* constraint which is satisfied when the (already known) optimum solution cost is reached. Therefore the Chapter 4 and Chapter 6 versions of these problems are equivalent in that a solution to one problem is the solution to the other. The situation is different for the over constrained timetabling problems, which have been made more difficult by adding additional soft constraints and so cannot be considered exactly equivalent to their CSP counterparts. Bearing this in mind, the Chapter 4 results for the nurse rostering problems show a similar algorithm ordering in that NOVELTY, RNOVELTY and TABU perform relatively worse than the weighting strategies. However if we compare the percentage of optimal solutions found in table 6.1 to the equivalent success rate in table 4.3 we find a proportionally greater deterioration in performance of the non-weighting algorithms. This is illustrated in table 6.4.

Rostering	MIN	GLS/UTIL	NOVELTY	RNOVELTY	TABU
CSP success rate	94.00	80.00	73.00	76.00	67.00
Over constrained optimal rate	76.25	30.63	21.25	21.88	9.38
Proportional change	1 : 1.23	1 : 2.61	1 : 3.44	1 : 3.47	1 : 7.14

Table 6.4. Comparison of Chapter 4 and Chapter 6 nurse rostering success rates

Similarly, comparing the timetabling success rates from Chapter 4 (*tt_rand* in table 4.3) to the over constrained success rates in table 6.2 shows another proportionally greater deterioration for the non-weighting algorithms (see table 6.5). In this case NOVELTY and RNOVELTY, which dominated the CSP timetable formulations, become clearly uncompetitive for the over constrained problems.

Timetabling	MIN	GLS/UTIL	NOVELTY	RNOVELTY	TABU
CSP success rate	95.00	95.00	100.00	97.00	91.00
Over constrained optimal rate	85.00	66.00	21.00	30.00	59.00
Proportional change	1 : 1.12	1 : 1.44	1 : 4.76	1 : 3.23	1 : 1.54

Table 6.5. Comparison of Chapter 4 and Chapter 6 timetabling success rates

In conjunction, the timetabling and nurse rostering results indicate that adding a two-tier weighting scheme (for hard and soft constraints) into a NOVELTY or TABU based algorithm causes a greater deterioration in performance than adding the equiva-

lent weighting scheme into our constraint weighting algorithm (note MIN uses the same fixed weight scheme as NOVELTY, RNOVELTY and TABU). This conclusion is supported by an intuitive understanding of the operation of the different algorithms: for TABU, NOVELTY and RNOVELTY adding weights to hard constraints will make the search topography more rugged, meaning the algorithms will have to search more extensively to escape from a local minimum, especially one that satisfies all hard constraints. The same situation is true of a weighting algorithm, except that weights will tend to build up on soft constraints that are difficult to satisfy, reducing the distinction between easy hard and difficult soft constraints, and so making the escape from a hard constraint satisfying minimum easier.

FWA versus DWA. In looking at the two dynamic constraint weighting schemes (FWA and DWA), FWA strongly dominates the timetabling problems, has a small advantage on the RLFAPs and is slightly worse than DWA for the nurse rostering problems. A likely explanation for these results is that FWA is better for longer term searches (the mean values tested for FWA were 889094 for timetabling, 1241642 for RLFAP and 294565 for nurse rostering). This better long term performance of FWA can be explained by FWA being able to make upward revisions to the overall weight of hard constraints in response to local conditions. DWA initially places greater importance on the hard constraints and only slowly reduces these weights. Therefore we would expect DWA to quickly find hard constraint satisfying solutions and, for easier problems, to drive more quickly to the optimal solution. However, if DWA is involved in a more protracted search it is likely to encounter new regions where the hard constraints are difficult to satisfy. Because it has no means to increase the overall importance of the hard constraints it must escape these regions by weighting hard constraints individually. In contrast, FWA, when encountering a region of difficult hard constraints, can easily shift the focus from the soft constraints by incrementing the overall weight of all hard constraints. In this way FWA can move more quickly to another hard constraint satisfying region and so should spend a greater proportion of the *later* stages of a search in promising (hard constraint satisfying) regions (in comparison to DWA).

Flexible versus Fixed Weighting. An interesting result of the study is that the dynamic weighting strategies have performed slightly better than the MinIncrement (MIN) algorithm. MinIncrement uses what is *probably* the best fixed increment (i.e. the optimal solution cost), a value that would typically be estimated from an analysis of the problem domain (as in Cha *et al*.'s study). In contrast, the dynamic weighting strategies do not rely on domain knowledge, and so avoid the effort and possible errors in using fixed increments, while delivering *at least* comparable performance.

GLS. It should be noted that the version of GLS developed in this study differs from the original proposed by [Voudouris and Tsang, 1996]. Not only do we not incorporate the Fast Local Search heuristic, we also use a fixed penalty increment and model the RLFAPs assignment costs differently. For comparison purposes we report in table 6.6 the published results for GLS [Voudouris and Tsang, 1995] with our results for the 4 RLFAP problems used in the study.

RLFAP Instance	Method	Best Solution	Average Cost (Std. Dev)	Average Time (CPU sec.)
scen01	GLS original	16	18.6 (2.3)	8.77
	GLS adapted	16	20.9 (4.8)	9.38
scen02	GLS original	14	14 (0.0)	0.59
	GLS adapted	14	14 (0.0)	0.97
scen03	GLS original	16	15.4 (1.3)	5.62
	GLS adapted	16	17.8 (5.9)	10.39
scen11	GLS original	28	n/a	98.97
	GLS adapted	38	43.2 (1.4)	10.96

Table 6.6. Comparison of original and adapted GLS performance

Table 6.6 shows the original GLS algorithm has generally better performance than our adaptation both in terms of average cost and standard deviation (note CPU time is not directly comparable as the experiments were performed on different machines). This indicates that Voudouris and Tsang's specialised modelling in conjunction with the Fast Local Search heuristic does produce better results on the RLFAPs. Therefore we should not conclude that FWA or DWA are better than GLS in any absolute sense. However we can say that within the common algorithmic and modelling framework chosen for our study, the GLS move selection heuristic did not do as well as FWA or

DWA. To investigate this further would require testing the FWA and DWA heuristics within the original GLS framework. We leave this for future research.

6.5 Summary

The main contributions of the Chapter are as follows:

- The development of a constraint weighting strategy that simulates the transformation of an over constrained problem with hard and soft constraints into an equivalent problem with a single constraint type, where the importance of each hard constraint is represented by repetition.

- The development of two dynamic constraint weighting strategies that adjust the number of repetitions of each hard constraint through dynamic feedback with the search space.
- The empirical evaluation of the new weighting strategies.

The main finding of the study is that for all the problem domains considered, one or other of the dynamic weighting strategies outperforms both the fixed weighting strategies and the alternative non-weighting strategies considered. Using comparisons with results in Chapter 4 we observe that constraint weighting performance is less degraded by the introduction of hard constraint weights than the alternative NOVELTY and TABU algorithms.

In Chapter 4 we concluded that (within our empirical study) constraint weighting performs better on problems where weighting can distinguish between groups of constraints. This led us to develop arc-weighting in Chapter 5 and then to investigate the distinction between hard and soft constraints in the current chapter. The superior performance of constraint weighting on over constrained problems therefore supports our original findings and further suggests constraint weighting is particularly suited to the over constrained problem domain.

Chapter 7

Conclusion

In this chapter we summarise the findings and contributions of the thesis and discuss future research directions.

7.1 Summary

The overall aim of the thesis has been to investigate the use of constraint weighting as a general purpose heuristic for constraint satisfaction. Addressing this aim, the broad conclusion of the thesis is that constraint weighting *is* a useful technique for constraint satisfaction, specifically for problems where constraint groups can be distinguished by the weighting process. Through the investigation we have also developed and empirically tested various improvements to constraint weighting, including arc weighting, hybrid weighting and dynamic constraint weighting techniques for hard and soft constraint problems.

In more detail, Chapter 2 started by placing constraint weighting in the context of the other major constraint satisfaction techniques and introduced a taxonomy of local search methods. Then in Chapter 3 we looked at modelling realistic problems within a general CSP framework. This led us to investigate the transformation of non-binary constraints to a dual graph binary representation and to propose a partial transformation model. In addition we looked at representing complex move operators and went on to develop an array-based domain representation and array-based resource constraints that internally represent and count domain value usage.

Chapter 4 introduced the empirical section of the thesis with an examination of the behaviour and application of constraint weighting in comparison with several other local search techniques and in relation to a range of CSPs and satisfiability problems. Through this analysis we developed a set of measures that describe problem structure and constraint weighting behaviour. These included constraint weight curves, which give a graphical picture of the distribution of weight across constraints and a constancy measure *Ct* that quantifies the amount of movement in the top 10% of weighted constraints. To further measure problem structure we examined the distribution of neighbour counts across variables. In conjunction and within our problem set, these measures indicated that constraint weighting does better on structured (as opposed to random) problems where it is able to distinguish between harder and easier groups of constraints. The empirical study also showed that constraint weighting is competitive with some of the best heuristics developed in the satisfiability domain (specifically NOVELTY and RNOVELTY) and that of three weighting heuristics examined none was a clear winner on all the domains considered. It was also observed that constraint weighting performance tends to decline faster on random problems as problem size grows (relative to the other techniques). This decline was explained by the falling rate of clustering between variables causing a corresponding fall in the probability of generating hard constraint groups. It was further observed that weighting techniques that add weight more infrequently tend to do better on larger problems. This led us to propose a weighting granularity effect, that causes the guidance of weighting to decline as the amount of weight and number of problem constraints grows (the work in Chapter 4 extends already published work in [Thornton and Sattar, 1999]).

Chapter 5 investigated whether constraint weighting can be improved as a general purpose CSP solving technique and introduced two main approaches: firstly the development of hybrid algorithms that combine weighting with the other local search heuristics introduced in Chapter 4, and secondly the introduction of an arc weighting algorithm that additionally weights the connections between constraints that are simultaneously violated at a local minimum. The arc weighting method built on the results of Chapter 4 by recognising and reinforcing the presence of harder sub-groups of constraints. Empirical results for the new methods indicated

that hybrid weighting algorithms are more likely to outperform their parent algorithms when solving problems for which the parent algorithms are fairly evenly matched. The arc weighting algorithm was shown to outperform a standard weighting algorithm on a range of CSPs and especially on problems the standard method found more difficult. However, arc weighting was found to be uncompetitive with the specialised algorithms developed for binary CSPs and satisfiability because it is unable to efficiently exploit the binary nature of these problems. We therefore concluded that arc weighting is best used as a general purpose technique for solving non-binary problems and as an 'add-on' to constraint weighting that is invoked in the later stages of a search (the work in Chapter 5 is based on ideas already proposed in [Thornton and Sattar, 1998a] and [Thornton and Sattar, 1999]).

Finally Chapter 6 looked at the application of constraint weighting to the domain of over-constrained problems with hard and soft constraints. This work again built on the findings of Chapter 4 which suggest that distinctions between constraint groups (such as hard and soft constraints) will favour a constraint weighting heuristic. Chapter 6 was specifically interested in developing a weighting scheme that can penalise frequently violated constraints without losing the original weight distinction between the hard and soft constraints in the system. To this end we proposed and empirically evaluated two constraint weighting heuristics that dynamically change the relative importance of the hard and soft constraint groups during the search by means of simulated hard constraint repetition. Our results indicated that the dynamic constraint weighting methods have a clear advantage over fixed weighting schemes and other selected local search techniques when solving hard and soft constraint systems. Given the underpinning of these results with the results from Chapter 4, we concluded that constraint weighting appears especially suited to the over-constrained problem domain where different groups of constraints have different levels of importance (Chapter 6 extends and updates work originally published in [Thornton and Sattar, 1998b].

7.2 Future Work

As previously discussed, the main aim of the thesis has been to investigate constraint weighting as a *general purpose* method for constraint satisfaction. This places the work within the broader context of developing automated ways of solving CSPs (i.e. without the human intervention required to choose or develop an appropriate heuristic). To this end the thesis has shown (not surprisingly) that none of the local search methods we have considered is superior in all situations. However, we have also shown that there are certain features of problem structure and weighting behaviour that indicate where constraint weighting may perform better than our other techniques. This further suggests that through an automated analysis of problem structure we may be able to decide in advance which algorithm is best suited for a particular problem. We have started the work in this area for constraint weighting, but equally we could have considered those conditions for which a tabu or stochastic method is better suited. The systemmatic categorisation of such information on a broad range of problem domains and the development of accurate and reliable measures would go a long way towards developing fully automated and efficient problem solving techniques. In relation to this area there has already been relevant work in categorising search space topology [Frank *et al.*, 1997], measuring randomness using 'approximate entropy' [Hogg, 1998], categorising problems according to their cost distributions [Gomes *et al.*, 1998] and also [Kwan, 1997]'s work on mapping CSPs to solution methods.

Another area related to our work is the development of algorithms that can dynamically change their search heuristic through feedback about performance during the search. This is connected to earlier work by [Minton, 1996] on MULTI-TAC and more recently to [Boyan and Moore, 1998]'s work on STAGE. Our method of measuring the consistency of membership of the top 10% of weighted constraints already suggests that if this consistency measure is low then constraint weighting is unlikely to do well. A next step would be to develop an algorithm that tracks consistency and is able to change heuristics if the consistency falls below a certain level. Another simple approach suggested by our work is to abandon the use of weights if a solution is not found relatively early during a search in relation

to the size of the problem (the definition of when ‘early’ would be for a particular problem is another research issue).

Following on from the WSAT approach of solving CSPs by selecting a violated constraint and trying to improve that constraint according to a given heuristic, we envisage building systems where different constraints use different heuristics on the *same* problem. Further, through a process of feedback during the search a constraint would be able to change or adapt its heuristic to suit local conditions. This idea is part of a larger plan to develop local search strategies that are controlled by the local decisions of autonomous constraint ‘agents’ (joining together the ideas of [Hogg and Williams, 1993] and [Lui and Sycara, 1995]). Here each agent would be able to identify and appropriately respond to the prevailing conditions in its immediate environment. There seems to be a close connection between the concept of an agent and the idea of an autonomous constraint. Such a constraint would have plans in the form of heuristics and rules for their application, beliefs about the environment (e.g. the instantiation of variables and the state of other constraints in the system) and a range of possible actions (e.g. changing the instantiation of the variables under its control). Seen this way a local search can already be understood as an emergent behaviour (i.e. solving the problem) based on a series of local decisions. Our idea is to extend the range of behaviours available to individual constraints with the objective of developing self-adapting systems that can learn the best way to solve a particular problem.

On a more immediate level there are several avenues to extend our existing work on constraint weighting. Firstly, we have concentrated on the *average* performance of the algorithms considered. Recent work [Hoos and Stutzle, 1999] has looked at categorising the run length distributions (RLDs) for various local search algorithms on *individual* problem instances. This work has shown that reporting the averaged mean and standard deviation over a number of problem instances does not necessarily present a complete picture of algorithm performance. Recognising the type of RLD and whether the RLDs of different algorithms cross gives further insight into whether one algorithm strictly dominates another and whether a restart strategy will benefit a particular approach. Extending this work to examine constraint weighting would both clarify the differences observed between techniques

in the thesis and provide guidance in the use of constraint weighting restart strategies. Also, we have only briefly looked at the area of hybrid local search techniques. Other work [Wu and Wah, 1999] has successfully applied a more sophisticated tabu and weighting algorithm to the larger parity learning and tower of hanoi problems from the DIMACS benchmark. Further investigation into constraint weighting hybrids has therefore already proved useful. As previously suggested, [Frank, 1997]'s weight decay scheme and [Voudouris and Tsang, 1996]'s GLS method both appear promising for larger problems and suggest a more detailed investigation of this area is required. Of all our results, the strong performance of the dynamic weighting schemes on over-constrained hard and soft constraint problems appears the most promising. It would therefore be worthwhile to extend our empirical study to see if the encouraging results are replicated in other domains. Finally, our work has already shown different weight heuristics perform differently on different problems. However, beyond recognising that UTILWGT does better on longer term searches (because it applies less weight) an explanation for the variations in weighting heuristic performance has not been proposed. A more detailed investigation into this question therefore also seems called for.

Appendix

Zero-One Block Constraints

Block constraints apply to staff members and student groups in the timetabling problem and specify that each staff member/student group should not be scheduled more than b consecutive class timeslots. These constraints can be modelled in a zero-one variable framework (introduced in Chapter 3) by building vectors to represent all $b + 1$ consecutive time slots for a group or staff member and constraining the total sum of these elements to be $\leq b$. For example, for staff member s, if $b = 4$ we would have constraints for all X_{ijk} taught by s where $k \geq 1$ and $k \leq 5$, $\Sigma X_{ijk} \leq 4$, and for $k \geq 2$ and $k \leq 6$, $\Sigma X_{ijk} \leq 4$, and so on for all valid time slot sequences. Then, if a particular X_{ijk} is changed we would require at most $b + 1$ vector sum evaluations to check the block for a particular staff member or student group, as shown in the following example:

Consider changing $X_{1,1,8}$ from 0 to 1 (which represents putting class 1 in room 1 at timeslot 8). Assuming staff member s who teaches class 1 must not be scheduled more than $b = 4$ consecutive timeslots, we must check 4 timeslots before and after timeslot 8 to verify the block constraint. Converting this into our vector representation, we must check all sequences of 5 timeslots that contain timeslot 8, i.e. $k \geq 4$ to $k \leq 8$, $k \geq 5$ to $k \leq 9$, $k \geq 6$ to $k \leq 10$, $k \geq 7$ to $k \leq 11$ and $k \geq 8$ to $k \leq 12$. Hence we make at most $b + 1 = 5$ evaluations.

The zero-one block constraint only *partly* represents a true block constraint because a local search of the zero-one representation allows the same class to be scheduled

more than once. Hence the block sum could be violated because a class is scheduled twice and not because the block length has been exceeded. This will not affect the correctness of a final solution (because other constraints will ensure a class is only scheduled once) but it will affect the guidance of a local search through non-feasible space.

Bibliography

[Abramson, 1992] D. Abramson. A very high speed architecture for simulated annealing. *IEEE Computing*, May:27-36, 1992

[Asahiro *et al.*, 1993] Y. Asahiro, K. Iwama, and E. Miyano. Random generation of test instances with controlled attributes. In *Proceedings of the 2nd DIMACS Challenge Workshop,* 1993.

[Bacchus and van Beek, 1998] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98),* pages 311-318, 1998.

[Battiti, 1995] R. Battiti. Reactive search: toward self-tuning heuristics. In *Proceedings of Applied Decision Technologies,* pages 1-19, 1995,

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

[Boyan and Moore, 1998] J. Boyan and A. Moore. Learning evaluation functions for global optimization and boolean satisfiability, In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98),* pages 3-10, 1998.

[Bowen and Dozier, 1996] J. Bowen, and G. Dozier. Constraint satisfaction using a hybrid evolutionary hill-climbing algorithm that performs opportunistic arc and

path revision. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, pages 326-331, 1996.

[Castell and Cayrol, 1997] T. Castell and M. Cayrol. Hidden gold in random generation of SAT satisfiable instances. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 372-377, 1997.

[Cha and Iwama, 1995] B. Cha and K. Iwama. Performance Test of Local Search Algorithms Using New Types of Random CNF Formulas. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 304-310, 1995.

[Cha and Iwama, 1996] B. Cha and K. Iwama. Adding new clauses for faster local search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, pages 332-337, 1996.

[Cha *et al.*, 1997] B. Cha, K. Iwama, Y. Kambayashi and S. Miyazaki. Local search algorithms for partial MAXSAT. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)*, pages 332-337, 1997.

[Connolly, 1992] D. Connolly. General purpose simulated annealing. *Journal of Operational Research Society*, 43(5):495-505, 1992.

[Davenport *et al.*, 1994] A. Davenport, E. Tsang, C. Wang, and K. Zhu. GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 325-330, 1994.

[Dantzig, 1963] G. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, N.J., 1963.

[Dechter, 1992]. R. Dechter. Constraint Networks. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 276-285, Wiley, New York, 1992.

[Frank, 1996] J. Frank. Weighting for Godot. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, pages 338-343, 1996.

[Frank, 1997] J. Frank. Learning Short-Term Weights for GSAT. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)*, pages 384-389, 1997.

[Frank *et al.*, 1997] J. Frank, P. Cheeseman, and J. Stutz. When gravity fails: local search topology. *Journal of Artificial Intelligence Research*, 7:249-281, 1997.

[Freuder and Hubbe, 1995] E. Freuder and P. Hubbe. Extracting constraint satisfaction subproblems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 548-555, 1995.

[Freuder and Wallace, 1992] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21-70, 1992.

[Gaschnig, 1977] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI'77)*, page 457, 1977.

[Gaschnig, 1978] J. Gaschnig. Experimental case studies of backtrack vs. Walz-type vs. new algorithms for satisficing-assignment problems. In *Proceedings of the 2nd National Conference of the Canadian Society for Computational Studies of Intelligence*, 1978.

[Gendreau *et al.*, 1994] M. Gendreau, A. Hertz, and G. Laporte. A tabu search heuristic for the vehicle routing problem. *Management Science*, 40(10):1276-1290, 1994.

[Gent and Walsh, 1993] I. Gent and T. Walsh. Towards an Understanding of Hill-climbing Procedures for SAT. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI'93)*, pages 28-33, 1993

[Gent *et al*. 1999] I. Gent, H. Hoos, P. Prosser, and T. Walsh. Morphing: combining structure and randomness. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99)*, 1999.

[Glover, 1989] F. Glover. Tabu search - part 1. *ORSA Journal on Computing*, 1(3):190-206, 1989

[Glover, 1990] F. Glover. Tabu search - part 2. *ORSA Journal on Computing*, 2(1):4-32, 1990.

[Gomes *et al*., 1998] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 431-437, 1998.

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

[Hertz *et al.*, 1995] A. Hertz, E. Taillard, and D. de Werra. *A Tutorial on Tabu Search.* Technical Report, Dept. de Mathematiques, MA-Ecublens, Lausanne, 1995.

[Hogg, 1998] T. Hogg. Which search problems are random? In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 438-443, 1998.

[Hogg and Williams, 1993] T. Hogg and C. Williams. Solving the Really Hard Problems with Cooperative Search. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI'93)*, pages 231-236, 1993.

[Hoos and Stutzle, 1999] H. Hoos and T. Stutzle. On the empirical evaluation of Las Vegas algorithms. In *Proceedings of the Workshop on Empirical Artificial Intelli-*

gence, 16th International Joint Conference on Artificial Intelligence, Stockholm, Sweden, 1999.

[Jiang *et al*., 1995] Y. Jiang, H. Kautz, and B. Selman. Solving problems with hard and soft constraints using a stochastic algorithm for MAX-SAT. In *Proceedings of the 1st International Joint Workshop on Artificial Intelligence and Operations Research*, Timberline, Oregon, 1995.

[Kumar, 1992] V. Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, Spring:32-43, 1992.

[Kwan, 1997] A. Kwan. *A framework for mapping constraint satisfaction problems to solution methods*, PhD Thesis, Department of Computer Science, University of Essex, U.K., July, 1997.

[Lo and Bavarian, 1992] Z. Lo and B. Bavaraian. Optimization of job scheduling on parallel machines by simulated annealing algorithms, *Expert Systems with Applications*, 4:323-328, 1992.

[Lui and Sycara, 1995] J. Lui and K. Sycara. Emergent constraint satisfaction through multiagent coordinated interaction. In C. Castelfranshi and J. Muller, editors, *From Reaction to Cognition*, pages 107-121, Springer, 1995.

[Mackworth, 1977] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99-118, 1977.

[Mackworth, 1987] A. Mackworth. Constraint Satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205-211, Wiley, New York, 1985.

[Marriott and Stuckey, 1998] K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[Mazure *et al.*, 1997] B. Mazure, S. Lakhdar, and E. Gregoire. Tabu search for SAT. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)*, pages 281-285, 1997.

[McAllester *et al.*, 1997] D. McAllester, B. Selman and H. Kautz. Evidence for invariants in local search. *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 321-326, 1997.

[Minton *et al.*, 1992] S. Minton, M. D. Johnston, A. B. Philips and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161-205, 1992

[Minton, 1996] S. Minton. Automatically configuring constraint satisfaction programs: a case study. *Constraints*, 1(1-2):7-43, 1996.

[Mitchell *et al.*, 1992] D. Mitchell, B. Selman and H. Levesque. Hard and easy distributions of SAT problems. *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'98)*, pages 459-465, 1992.

[Mitchell, 1998] D. Mitchell. Hard problems for CSP algorithms. *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 398-405, 1998.

[Morris, 1993] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI'93)*, pages 40-45, 1993.

[Papadimitriou, 1994] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Papadimitriou and Steiglitz, 1982] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[Poole *et al.*, 1998] D. Poole, A. Mackworth and R. Goebel. *Computational Intelligence: a Logical Approach*. Oxford University Press, Oxford, 1998.

[Prosser, 1996] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1-2):81--111, March 1996

[Rossi *et al.*, 1990] F. Rossi, C. Petrie, and V. Dhar. On the Equivalence of Constraint Satisfaction Problems, In *Proceedings of the European Conference on Artificial Intelligence (ECAI'90)*, pages 550-556, 1990.

[Sabin and Freuder, 1997] D. Sabin and E. Freuder. Understanding and improving the MAC algorithm. In *Proceedings of the 3rd International Conference on the Principles and Practice of Constraint Programming (CP'97)*, Springer, 1997.

[Schaerf, 1996] A. Schaerf. Tabu Search Techniques for Large High-School Timetabling Problems. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 363-368, 1996.

[Selman and Kautz, 1993] B. Selman and H. Kautz. Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 290-295, 1993.

[Selman *et al.*, 1992] B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 440-446, 1992.

[Selman *et al.*, 1994] B. Selman, H. Kautz, and B. Cohen. Noise strategies for local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 337-343, 1994.

[Selman *et al.*, 1997] B. Selman, H. Kautz, and D. McAllester. 1997. Ten Challenges in Propositional Reasoning and Search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 50-54, 1997.

[Stergiou and Walsh, 1999] K. Stergiou and T. Walsh. Encodings of non-binary constraint satisfaction problems. In *Proceedings of Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99)*, pages , 1999.

[Tabachnick and Fidell, 1989] B. Tabachnick and L. Fidell. *Using Multivariate Statistics*. Harper Collins, New York, 1989.

[Thornton, 1995] J. Thornton. *An enhanced cyclic descent algorithm for nurse rostering*. Honours Thesis, Faculty of Engineering and Applied Science, Griffith University Gold Coast, Australia, 1995.

[Thornton and Sattar, 1996] J. Thornton and A. Sattar. An integer programming-based nurse rostering system. In *Proceedings of the 2nd Asian Computing Science Conference (ASIAN '96)*, pages 357-358. Springer, 1996.

[Thornton and Sattar, 1997] J. Thornton and A. Sattar. Applied partial constraint satisfaction using weighted iterative repair. In *Proceedings of the 10th Australian Joint Conference on Artificial Intelligence (AI'97)*, pages 57-66. Springer, 1997.

[Thornton and Sattar, 1998a] J. Thornton and A. Sattar. Using arc weights to improve iterative repair. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 367-372, 1998.

[Thornton and Sattar, 1998b] J. Thornton and A. Sattar. Dynamic constraint weighting for over-constrained problems. In *Proceedings of the 5th Pacific Rim International Conference on Artificial Intelligence (PRICAI-98)*, pages 377-388, Springer, 1998.

[Thornton and Sattar, 1999] J. Thornton and A. Sattar. On the behaviour and application of constraint weighting. In *Proceedings of the 5th International Conference on the Principles and Practice of Constraint Programming (CP'99),* pages 446-460. Springer, 1999.

[Voudouris and Tsang, 1995] C. Voudouris and E. Tsang. *Function Optimization using Guided Local Search.* Technical Report CSM-249, Department of Computer Science, University of Essex, U.K. September 1995.

[Voudouris and Tsang, 1996] C. Voudouris and E. Tsang. Partial Constraint Satisfaction Problems and Guided Local Search. In *Proceedings of Practical Application of Constraint Technology (PACT'96),* pages 337-356, 1996.

[Wah and Shang, 1997] B. Wah and Y. Shang. Discrete Lagrangian-based search for solving MAX-SAT problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97),* pages 378-383, 1997.

[Wallace and Freuder, 1995] R. Wallace and E. Freuder. Anytime Algorithms for Constraint Satisfaction and SAT Problems. *SIGART Bulletin,* 7(2):7-10, 1995.

[Walser *et al.,* 1998] J. Walser, R. Iyer and N. Venkatasubramanyan. An integer local search method with application to capacitated production planning. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98),* pages 373-379, 1998.

[Walsh, 1999] T. Walsh. Search in a small world. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99),* pages 1172-1176, 1999.

[Walsh, 2000] T. Walsh. Reformulating propositional satisfiability as constraint satisfaction. *Proceedings of SARA-2000* (to appear), Springer-Verlag, 2000.

[Warner, 1976] D. Warner. Scheduling nursing personnel according to nursing preference: A mathematical programming approach. *Operations Research* 24(5):842-856, 1976.

[Watts and Strogatz, 1998] D. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440-442, 1998.

[Wu and Wah, 1999] Z. Wu and B. Wah. Trap escaping strategies in discrete Lagrangian methods for solving hard satisfiability and maximum satisfiability problems. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99)*, 1999.

[Yugami *et al*., 1994] N. Yugami, Y. Ohta, Y., and H. Hara. Improving repair-based constraint satisfaction methods by value propagation. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 344-349, 1994.