Dynamic Constraint Weighting for Over-Constrained Problems

John Thornton[1] and Abdul Sattar[2]

[1]School of Information Technology, Griffith University Gold Coast
Parklands Drive, Southport, Qld 4215, Australia
j.thornton@eas.gu.edu.au

[2]School of Computing and Information Technology, Griffith University
Kessels Road, Nathan, Qld, 4111, Australia
sattar@cit.gu.edu.au

Abstract. Many real-world constraint satisfaction problems (CSPs) can be over-constrained but contain a set of mandatory or *hard* constraints that *have* to be satisfied for a solution to be acceptable. Recent research has shown that constraint weighting local search algorithms can be very effective in solving a variety of CSPs. However, little work has been done in applying such algorithms 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 paper 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, but the hard constraints have been repeated. 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. The dynamic constraint weighting algorithms are compared with two algorithms that maintain a fixed number of hard constraint repetitions, using a test bed of over-constrained timetabling and nurse rostering problems. The results show the dynamic strategies outperform both fixed repetition approaches.

1 Introduction

Recent research has shown that local search techniques can be remarkably effective in solving certain classes of Constraint Satisfaction Problem (CSP) [10]. Particular interest has focussed on the use of GSAT and variants to solve Conjunctive Normal Form (CNF) satisfiability problems [7]. Attempts to improve GSAT have lead to the development of a new class of clause weighting algorithms [6]. These algorithms escape local minimum situations by adding weights to unsatisfied clauses. Further research has looked at applying constraint weights to more general CSPs [8], and at

improving the weighting strategy by weighting the *connections* between constraints [9]. Cha et al. [2] have also looked at using clause weighting in solving an over-constrained timetabling problem. This paper further investigates the use of weighted local search in solving over-constrained problems.

An over-constrained problem is defined as a standard CSP (ie 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 [10]. Given all constraints are of equal importance, a standard weighting algorithm can be applied to an over-constrained problem with minimal modification (see section 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 when escaping 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? The primary purpose of the paper is to answer this question.

Cha et al. [2] proposed an initial answer 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).

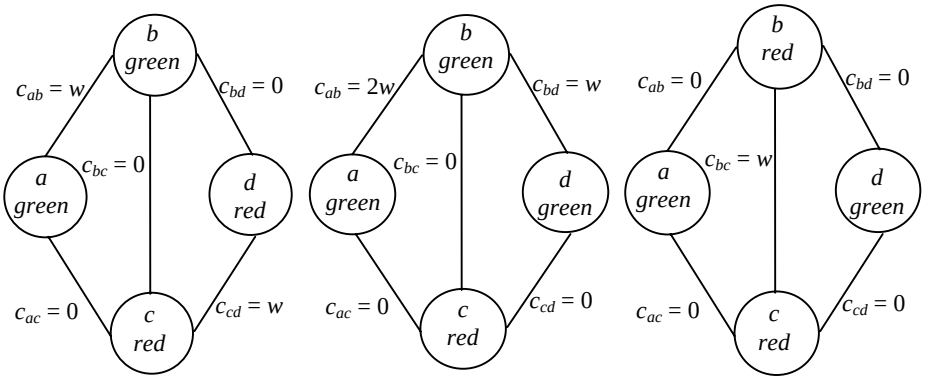
The motivation of the study is to encourage the implementation of local search weighting techniques developed in the domain of CNF satisfiability to the more complex ‘real-world’ of constraint satisfaction. The dynamic weight incrementing algorithms allow constraint weighting to be efficiently applied to over-constrained systems that have *already* been weighted. The remainder of the paper describes the algorithms used in the study, followed by an experimental evaluation of constraint weighting based on a test-bed of real-world staff scheduling and timetabling problems.

2 Constraint Weighting Algorithms

Constraint weighting algorithms are extensions of local search or iterative repair. An iterative repair algorithm starts with an initial instantiation of all problem variables and attempts to improve this solution by repeatedly selecting the ‘best’ improving local move. The algorithm terminates when no further improving move can be found. This simple approach proves very effective for many problem domains (for instance see [5]). However, a local search frequently terminates on a non-optimal solution or

local minimum, and a method is required to escape and continue the search. A simple approach is to restart the algorithm from a different initial position. GSAT accepts moves of equal cost in an attempt to find a better move later in the search. Other approaches include randomly selecting a move (as in simulated annealing and random walk [7]) and selecting the best move that does not repeat a previous move (as in tabu search [4]). Constraint weighting takes the approach of adding a weight to all violated constraints in a local minimum [6]. This changes the cost surface of the problem until a lower cost solution becomes accessible. This is illustrated in the following example:

Example. Consider the over-constrained graph colouring problem in figure 1. The nodes *a*, *b*, *c* and *d* represent the variables or areas to be coloured, each having a two value domain {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. Using a simple cost function, such that each constraint violation adds a cost of w to the solution, the situation in figure 1(a) represents a local minimum of cost $2w$. A constraint weighting algorithm would continue by adding a further weight w to each violated constraint until the cost of the solution becomes $4w$. This alters the problem so that a choice of lower cost moves become available. Figure 1(b) 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 $2w$. From 1(b) the best cost decreasing move is to change *b* to red, leading to the (optimal) solution in 1(c) where only one constraint, c_{bc}, is violated:



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

Fig. 1: Graph Colouring CSP

Figure 2 gives the pseudocode for the basic constraint weighting strategy used in the study. As the algorithm solves over-constrained problems, it needs to keep track of the best solution currently found in the search. This is not required for standard CSPs because a clear stopping condition exists (ie when *all* the constraints are satisfied). However, 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, hence the need to store each successive best solution as it is found.

In addition, 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 and use this measure to evaluate the best solution.

```
procedure WeightedIterativeRepair
begin
    CurrentState ← set variables to initial assignments
    BestCost ← UnweightedCost(CurrentState), BestState ← CurrentState, StuckCounter ← 0
    while UnweightedCost(BestState) > DesiredCost and StuckCounter < MaxStucks do
        if CurrentState is not a local minima then
            for each variable v_i involved in a constraint violation
                for each move m_j in the domain of v_i
                    if UnweightedCost(CurrentState + m_j) < BestCost then
                        BestState ← CurrentState + m_j
                        BestCost ← UnweightedCost(CurrentState + m_j)
                    if WeightedCost(CurrentState + m_j) < WeightedCost(CurrentState) then
                        CurrentState ← CurrentState + m_j
                end for
            end for
        else
            IncreaseViolatedConstraintWeights()
            StuckCounter ← StuckCounter + 1
    end while
end
```

Fig. 2. The Weighted Iterative Repair Algorithm

2.1 Weighting with Hard and Soft Constraints

Constraint weighting was developed as an enhancement to GSAT for CNF problems. Empirical studies have shown clause weighting to be one the best approaches for CNF satisfiability [1] [3] [7]. Further work has looked at applying constraint weighting to more general CSPs such as nurse rostering [8], university timetabling [2], graph colouring and blocks world planning [7]. Several enhancements to weighting have also been proposed, including causing weights to decay over time [3], adding new clauses for CNF problems [1] and weighting connections between constraints [9].

Constraint weighting is therefore recognised as an important and 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 [2]. 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 [8] 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 algorithms attempt to satisfy as many as possible soft constraints 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 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 study is the extension of Cha et al.'s concept of *repeating* hard constraints [2]. 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, 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 our 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 3: *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 constraints, 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 3(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 3(c)), whereas in the new constraint weighting scheme w_A and w_B will each be incremented by $n \times w$ to 6 (figure 3(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.

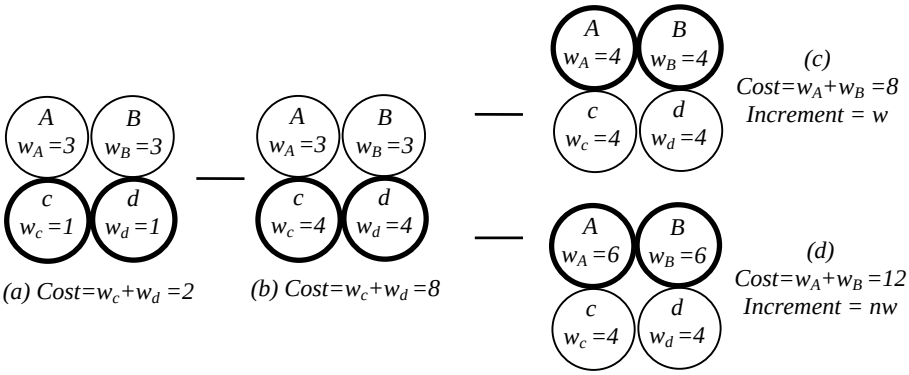


Fig. 3. Weighting Hard and Soft Constraints

Deciding the Initial Hard Constraint Weights. Cha et al. [2] 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 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.

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 paper 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 (ie the maximum value). Then, as soon as a solution is found where all hard constraints are satisfied (ie an *acceptable* solution), the value of *n* is adjusted downwards to equal the number of soft constraints currently violated (s_{cur}). Each time a new acceptable solution is found such that $s_{cur} < n$, then *n* is set to s_{cur} (the new best level of soft constraint violation), ie 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:

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

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

(Note that n is equivalent to *BestCost* in figure 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 (ie 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 4 gives the pseudocode necessary to implement FWA (Note IncreaseViolatedConstraintWeights() is called from the main constraint weighting algorithm in figure 2).

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

Fig. 4. The Flexible Weight Adjustment Algorithm

3 Experiments

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 2.2). An implementation of Cha et al.'s reset algorithm [2] 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.

3.2 Test Problems

The algorithms were tested on a set of 16 over-constrained nurse rostering problems taken from real situations in a Queensland public hospital. The rostering problem involves allocating a set of pre-generated legal schedules to each nurse in a roster, such that all hard constraints involving numbers of staff for each shift are satisfied. The soft constraints define how attractive a schedule is for a nurse. A typical problem involves 25-35 nurses, each with up to 5,000 legal schedules, and approximately 400 constraints. Further details of the problems are described elsewhere [8]. 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 [8].

The second over-constrained problem was taken from another real-life situation of university timetabling. A single, large problem was considered involving 1237 classes, 287 full-time and part-time staff, 103 rooms and 1511 student groups of 1 to 5 students. In this problem the hard constraints are set to avoid timetable clashes and the soft constraints define the preferred class times for staff members and student groups. The problem proved too large to solve in total, so it was divided into 4 sub problems: firstly laboratories are allocated, then lectures, then tutorials and finally the remaining classes. The results reported here refer to solution times and constraint satisfaction levels for phase two of the problem (laboratories + lectures).

3.3 Results

All problems were solved on a Sun Creator 3D-2000. For the rostering problems, the algorithms were either terminated on finding an optimum solution, or after 250 local minimum were encountered without improvement. The timetable problem, being

much larger and with no known optimal solution, was terminated after 75 phase two iterations of the main program loop. The optimum timetabling solution was then defined as the best solution found in all runs.

Table 1 shows the average times and proportion of problems solved for seven runs of each test problem with each algorithm. In all cases the averages are calculated only for those runs that actually found an optimal solution. The results show the Flexible Weight Adjustment algorithm (FWA) has the best overall performance on both problems (ie it equals or exceeds the other algorithms in the proportion of optimal solutions found and has the smallest average execution times). The results also show the MaxIncrement (MAX) algorithm is unable to reliably find optimal solutions (only 51% of roster and 29% of timetable runs were successful), and generally has longer execution times for those problems it can solve.

	Roster Results				Timetable Results			
Method	FWA	MIN	MAX	DWA	FWA	MIN	MAX	DWA
Mean time (secs)	141.3	169.6	306.7	159.5	674	1192	1163	900
% Optimal	79.5	78.6	50.9	76.8	86	86	29	86
% Unsolved	0.9	2.7	0.9	0	0	0	0	0

Table 1. Proportions and Average Solution Times for each Method and Problem

Overall, the dynamic weighting strategies (FWA and DWA) proved more efficient at finding optimal solutions than the fixed weighting strategies (MIN and MAX). However, an important element in evaluating local search is the solution path represented by the so-called 'anytime curve' [10]. This plots the cost of the best solution found in the search against execution time, and represents 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. Figures 6 and 7 show the anytime curves for each method and problem type. In these graphs, the y-axis represents the sum of all soft constraint costs of the best solutions found at a given time for all runs of an algorithm.

4 Analysis

Figures 6 and 7 show the evaluation of the algorithms is more complex than a simple comparison of execution times. There is little to distinguish each algorithm in the timetable problem curves (figure 6), but these results are for a *single* problem, repeatedly solved, hence it would be unwise to generalise. With the roster curves in figure 7 (using 16 problem instances), the DWA curve is noticeably lower than the other curves. Although DWA's ability to find an optimal solution is slightly inferior to FWA, the faster descent of DWA indicates it may be more useful when looking for 'good enough' solutions. FWA is more appropriate for longer searches where small cost improvements are considered important. These results can be inferred from the

algorithms directly: DWA initially places greater importance on the hard constraints and only slowly reduces these weights. Therefore we would expect DWA to quickly find acceptable solutions of reasonable quality. Then, as DWA approaches an optimum it will find it harder to move by violating a hard constraint, because the relative weights of the hard and soft constraints are only adjusted when a new best solution is found. In contrast, FWA starts by strongly valuing the soft constraints, and so finds acceptable solutions more slowly. However, as FWA's ability to adjust weights remains constant regardless of the distance from an optimum solution, we would expect FWA to be more effective in the later stages of the search.

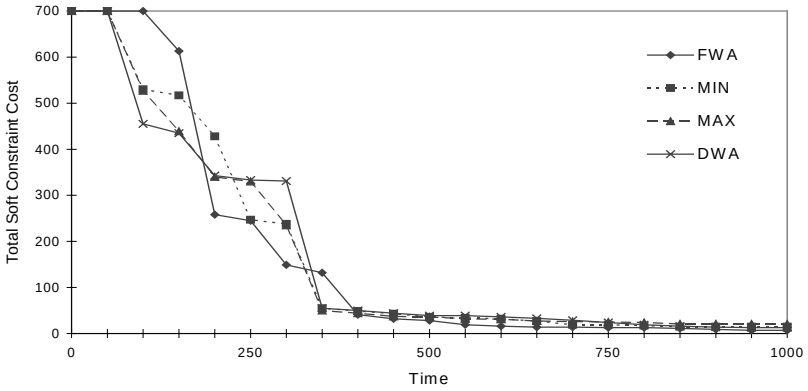


Fig 6. Anytime Curves for the Timetabling Problem

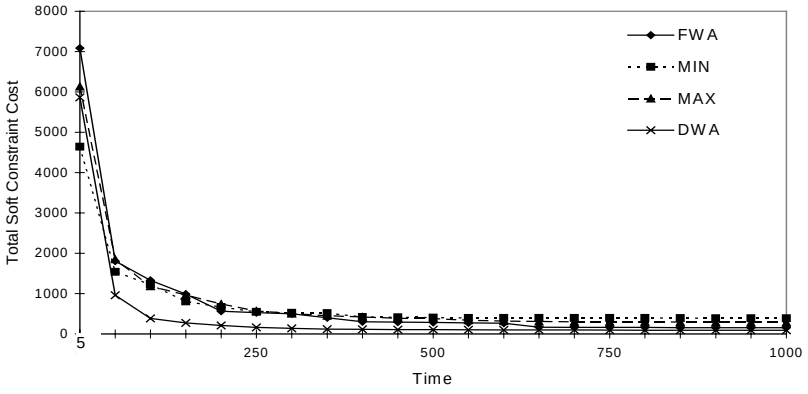


Fig 7. Anytime Curves for the Roster Problems

An interesting result of the study is that both flexible weighting strategies have performed slightly better than the MinIncrement (MIN) algorithm. MinIncrement uses what is *probably* the best fixed increment (ie 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.

5 Conclusions and Further Work

The main contributions of the paper 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 both dynamic weighting strategies outperform alternative fixed weighting strategies on a test bed of over-constrained timetable and nurse rostering problems. Clearly further empirical work is required to evaluate proposed algorithms. Future research will look at timetabling in more detail and at solving random over-constrained CSP and CNF problems. We will also look at comparing dynamic constraint weighting with other local search techniques such as WSAT, simulated annealing and tabu search.

References

1. B. Cha and K. Iwama. Adding new clauses for faster local search. In *Proc. of AAAI-96*, pages 332-337, 1996.
2. B. Cha, K. Iwama, Y. Kambayashi and S. Miyazaki. Local search algorithms for partial MAXSAT. In *Proc. of AAAI-97*, pages 332-337, 1997.
3. J. Frank. Learning short-term weights for GSAT. In *Proc. of AAAI-97*, pages 384-389, 1997.
4. F. Glover. Tabu search - part 1. *ORSA J on Computing*, 1(3):190-206, 1989.
5. S. Minton, M. D. Johnston, A. B. Philips and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artif. Intell.*, 58:161-205, 1992.
6. P. Morris. The breakout method for escaping local from minima. In *Proc. of AAAI'93*, pages 40-45, 1993.
7. B. Selman and H. Kautz. Domain independent extensions to GSAT: Solving large structured satisfiability problems. In *Proc. of IJCAI'93*, pages 290-295, 1993.
8. J. R. Thornton and A. Sattar. Applied partial constraint satisfaction using weighted iterative repair. In A. Sattar editor, *Advanced Topics in Artificial Intelligence*, pages 57-66. Springer-Verlag, 1997.
9. J. R. Thornton and A. Sattar. Using arc weights to improve iterative repair. In *Proc. of AAAI '98*, (to appear), 1998.
10. R. J. Wallace and E. C. Freuder. Heuristic methods for over-constrained constraint satisfaction problems. In M. Jampel, E. Freuder and M. Maher, editors, *Over-Constrained Systems*, pages 207-216. Springer-Verlag, 1996.