Using Arc Weights to Improve Iterative Repair

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

One of the surprising findings from the study of CNF satisfiability in the 1990's has been the success of iterative repair techniques, and in particular of weighted iterative repair. However, attempts to improve weighted iterative repair have either produced marginal benefits or rely on domain specific heuristics. This paper introduces a new extension of constraint weighting called Arc Weighting Iterative Repair, that is applicable outside the CNF domain and can significantly improve the performance of constraint weighting. The new weighting strategy extends constraint weighting by additionally weighting the connections or arcs between constraints. These arc weights represent increased knowledge of the search space and can be used to guide the search more efficiently. The main aim of the research is to develop an arc weighting algorithm that creates more benefit than overhead in reducing moves in the search space. Initial empirical tests indicate the algorithm does reduce search steps and times for a selection of CNF and CSP problems.

Introduction

One of the key findings in the study of Conjunctive Normal Form (CNF) satisfiability in the 1990's is that relatively simple iterative repair or local search techniques, like GSAT, are effective in finding answers to hard satisfiable CNF problems (Selman, Levesque and Mitchell 1992). A second finding is that the performance of such algorithms can be significantly improved by adding a simple clause or constraint weighting heuristic (Morris 1993, Selman and Kautz 1993, Cha and Iwama 1995). While research has been conducted into different implementations of weighting strategies (Frank 1996, 1997), the basic concept of constraint weighting has only been extended for solving CNF problems (Cha and Iwama 1996, Castell and Cayrol 1997). This paper is concerned with *domain independent* improvements to constraint weighting and is motivated by the success of constraint weighting in solving various Constraint Satisfaction Problems (CSPs) (eg Thornton and Sattar 1997). The basic question for all non-trivial iterative repair algorithms is how to escape from a local minimum (a local minimum being defined as a solution from which no single change of variable value can lead to an improved solution). GSAT's answer is to make a series of random 'sideways' moves (ie moves to equivalent cost solutions) until either a maximum number of moves have been performed or a reduced cost solution is found. If the maximum number of moves ('maxflips') is reached then the algorithm is re-started from a random point (Selman, Levesque and Mitchell 1992). In contrast, constraint weighting works by adding a weight to each constraint (clause) that is violated at a local minimum and then continuing the search. The weighting changes the shape of the cost surface so that another solution is eventually preferred and the local minimum is exceeded. The cost surface itself is determined by summing the weights of all violated constraints at each solution point (Morris 1993).

Frank (1996, 1997) suggested several performance enhancing modifications to the weighting algorithm, including updating weights after each move (instead of at each minimum), 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. 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) 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 paper we present an enhanced weighting algorithm that can be incorporated into a domain independent iterative repair approach. The motivation is to improve the performance of weighting not just in the CNF domain but to investigate weighting as a general technique for solving CSPs. Specifically, the paper looks at whether there is any benefit in weighting the connections or arcs between violated constraints. An important aim of the

Copyright © 1998, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

research is to develop an arc weighting algorithm where the benefit of a more informed search outweighs the computational cost of updating and calculating the arc weights. The new approach is called Arc Weighting Iterative Repair (AWIR) and is tested on a set of CNF 3SAT problems and on a set of CSP scheduling problems.

Arc Weighting Iterative Repair

The Arc Weighting Iterative Repair 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 (clauses) 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 1:

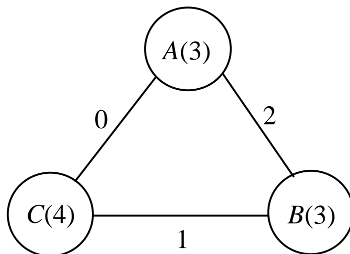

Figure 1: A Simple Constraint Weighting Scenario

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 (ie 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 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.

An Efficient Network Representation

The first step in representing the network graph is to realise 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 obviously changes according to the type of problem and the state of the search. Next arises the problem of how to represent and update the arc weights. This is done by first constructing an $n \times n$ 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 (ie 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 it's original state. This algorithm is shown in figure 2. Upon reaching a local minimum, the CList is reconstructed and the ArcArray counts updated accordingly (see figure 3).

```
procedure Move(CList, ArcArray, TCost, OldValue, NewValue)
begin
  CopyList ← CList, Improve ← False,
  Counter ← 0, Diff ← 0
  for each constraint c_i that contains OldValue do
    CChange ← cost change from OldValue to NewValue for c_i
    if CChange < 0 then Improve ← True
    if c_i already weighted and CChange <> 0 then
      add c_i to TList, Counter ← Counter + 1
    end if
    Diff ← Diff + CChange
  end for
  if Improve = True and Counter > 0 then
    for each constraint c_i in TList do
      if c_i violated with OldValue then
        if c_i satisfied with NewValue then
          delete c_i from CList
          for each constraint c_j in CList do
            Diff ← Diff - ArcArray[i][j](Cost(c_i) + Cost(c_j))
        end if
      else
        for each constraint c_j in CList do
          Diff ← Diff + ArcArray[i][j](Cost(c_i) + Cost(c_j))
        insert c_i into CList
      end if
    end for
  end if
  if Diff < 0 then accept NewValue, TCost ← TCost + Diff
  else if Diff = 0 then randomly accept NewValue
  if NewValue accepted then OldValue ← NewValue
  else CList ← CopyList
end
```

Figure 2: Move Selection Algorithm

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 3). 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. The overall arc weighting algorithm is shown in figure 3.

```
procedure ArcWeightingIterativeRepair
begin
  set variables to initial assignments, CList ← Empty
  TCost ← cost of initial solution,   ArcArray ← 0
  while unweighted cost of current solution > DesiredCost do
    if current state is not a local minimum then
      for each variable v_i involved in constraint violation do
        OldValue_i ← StartValue_i ← current value of v_i
        for each domain value d_j of v_i do
          if d_j <> StartValue_i then
            Move(CList, ArcArray, TCost, OldValue_i, d_j)
          end if
        end for
      end for
    else
      MoveSideways(), CList ← Empty
      for each constraint c_i in problem do
        if c_i currently violated then
          proportionally increment weight of c_i
          for each constraint c_j in CList do
            ArcArray[i][j] ← ArcArray[i][j] + 1
          end for
          add c_i to CList
        end if
      end for
      TCost ← cost of current solution
    end if
  end while
end
```

Figure 3: The Arc Weighting Iterative Repair Algorithm

Experiments

The Arc Weighting Iterative Repair (AWIR) algorithm was primarily developed as a means of solving general (non-binary) CSPs. It is designed to accept problems in the form of sets of variables with domains and sets of constraints between the variables. Consequently the CNF formulas used in the study were transformed to equivalent CSPs in the following way: a 3SAT clause is modelled as 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 $a_i x_i$ corresponds to the ith 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).

The Arc Weighting Iterative Repair algorithm is compared to the same algorithm with the arc weighting features switched off. This standard algorithm is referred to as Weighted Iterative Repair (WIR). The aim of the study is to find whether any benefit is obtained from arc weighting over simple weighting, so no further algorithms are considered (for a comparison of weighting to other techniques see Cha and Iwama 1995; Thornton and Sattar 1997). The two weighting algorithms are tested on a set of 3SAT CNF problems and a set of real-world staff scheduling CSPs. Two classes of CNF problem are used:

- randomly generated 3SAT 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 *r*100 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 Cha and Iwama 1995). These problems are prefixed an *o* followed by the number of variables (as above).

The scheduling 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 allowable 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.

Performance Measures

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 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 3), Hills is the mean number of *improving* moves made by the algorithm and Minima is the mean number of local minima encountered. The staff scheduling problems are reported as ss.csp.

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 (eg Cha and Iwama 1996, Frank 1996), this measure was found to be inadequate for comparing AWIR and WIR. As table 1 shows, for several problems the number of hill climbing moves made by AWIR exceeds WIR, while the AWIR 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 (eg 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 1). However, the Loops measure does not capture the extra work done by the AWIR algorithm in maintaining the CList (see figure 2). For this reason, execution times are also recorded.

Problem	Method	Time	Std Dev	Loops	Hills	Minima
*r*100	WIR	7.43	10.39	3955	3035	1398
	AWIR	6.04	7.90	2354	4218	637
*r*200	WIR	56.27	97.57	17171	9723	6614
	AWIR	20.91	27.02	4654	10839	1198
*r*400	WIR	342.23	202.10	61534	47383	22566
	AWIR	79.34	69.42	10779	31879	2684
*o*100	WIR	9.89	4.59	13322	5797	5101
	AWIR	6.41	3.49	7908	5796	2531
*o*200	WIR	88.25	45.25	66072	23557	26274
	AWIR	56.74	38.32	40618	27879	13316
ss.csp	WIR	144.35	250.38	207	390	69
	AWIR	74.02	97.68	136	475	38

Table 1: Comparison of mean performance values

Problem	Time	Std Dev	Loops	Hills	Minima
*r*100	.81	.76	.60	1.39	.46
*r*200	.37	.28	.27	1.11	.18
*r*400	.23	.34	.18	.67	.12
*o*100	.65	.76	.59	1.00	.50
*o*200	.64	.85	.62	1.18	.51
ss.csp	.51	.39	.66	1.22	.55

Table 2: Table 1 AWIR values as a proportion of WIR values

Analysis

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 4 and 5).

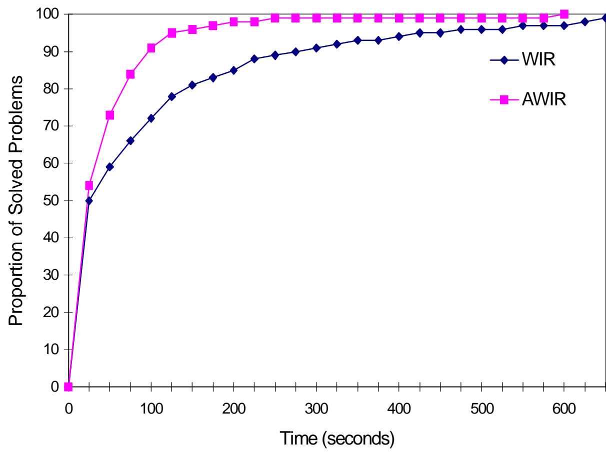

Figure 4: Proportion of solved problems by time

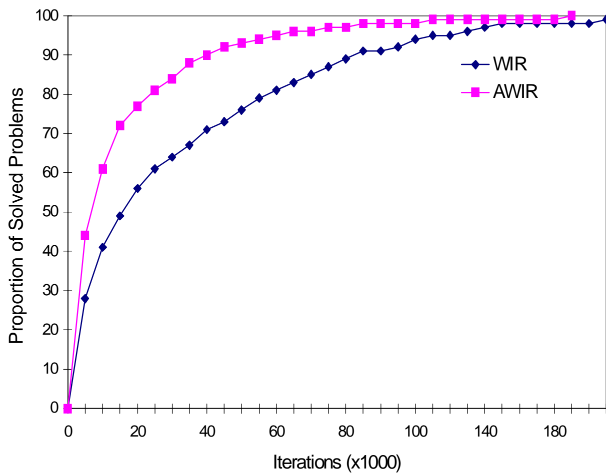

Figure 5: Proportion of solved problems by iterations

Distinguishing Moves

Table 2 re-expresses the results from table 1, giving the AWIR values as a proportion of the WIR values, and more clearly shows the relative differences between the algorithms. In all cases the AWIR results are less than the WIR results, except for the number of hill climbing or improving moves. The hill climb counts are shown in more detail in figure 6, 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 AWIR (see figure 7) indicate a more diverse search is occurring. More precisely, the hill climbing behaviour shows that, for a given number of iterations, AWIR is more likely to find a hill climbing move than WIR, 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.

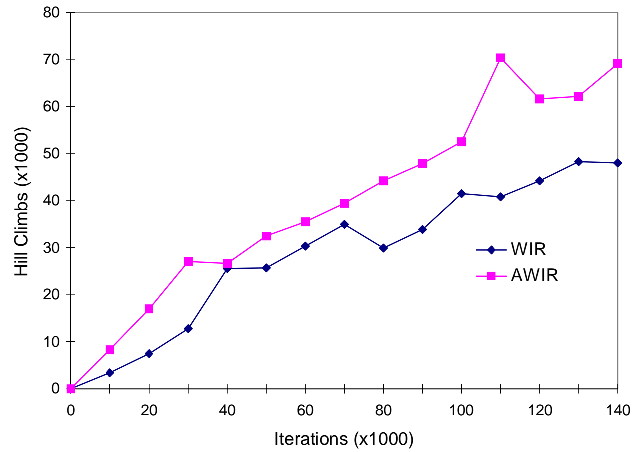

Figure 6: Comparison of hill climbing moves

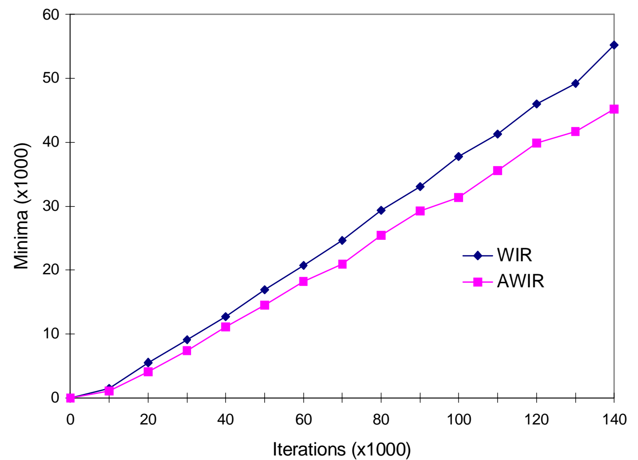

Figure 7: Number of minima by iterations

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 AWIR (compare figures 4 and 5). This reflects the cost to AWIR of using arc weights and is further analysed in table 3. Here the average number of iterations per second are calculated for each algorithm and problem class. The table shows the AWIR main loop is running at approximately 74% of the speed of WIR 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 1) and hence add to AWIR's overhead. However, a larger CList indicates that arc weights are also giving more guidance to the search, and so a counter-balancing improvement in search efficiency would be expected (as the results demonstrate).

Problem	Method	Loops/Time	AWIR/WIR
*r*100	WIR	532.30	73.22%
	AWIR	389.74	
*r*200	WIR	305.15	72.94%
	AWIR	222.57	
*r*400	WIR	179.80	75.56%
	AWIR	135.86	
*o*100	WIR	1347.02	91.59%
	AWIR	1233.70	
*o*200	WIR	748.69	95.61%
	AWIR	715.86	
ss.csp	WIR	1.43	92.03%
	AWIR	1.32	

Table 3: Comparison of Iteration Speed

Divergence

A further property of AWIR is that solution times tend to be more predictable or less divergent than for WIR. This is shown in the execution time standard deviations and in the graph of figure 8, 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.

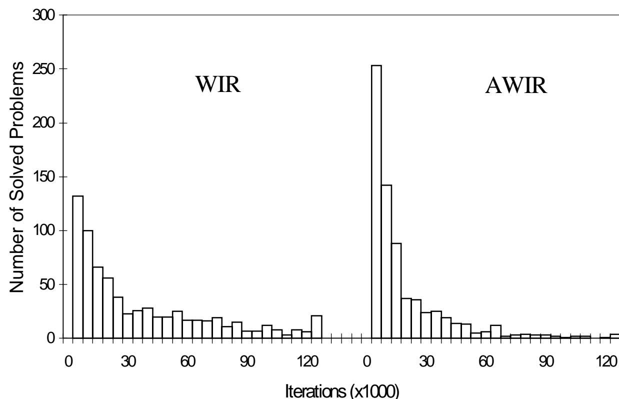

Figure 8: Number of solved CNF problems by iterations

Conclusions and Further Work

The main finding of this study is that arc weighing can improve the performance of a simple weighting algorithm on a range of different problems. While specialised algorithms like ANC may be more efficient for random CNF problems, arc weighting is domain independent and therefore is not tied to a particular problem formulation. This domain independence also means arc weighting can be used in conjunction with other techniques developed to improve simple weighting. The main technical contribution of the paper is the presentation of an efficient arc weighting algorithm, whose benefits have been shown to exceed the computational costs.

Ongoing research is concerned with a more extensive empirical evaluation of arc weighting and the integration of further performance enhancing strategies. The authors are also interested in applying weighting strategies in solving over-constrained problems.

Acknowledgments

The authors would like to acknowledge the assistance of Byungki Cha and Paul Morris in providing the CNF test instances used in this study.

References

Castell, T., and Cayrol, M. 1997. Hidden Gold in Random Generation of SAT Satisfiable Instances. In Proceedings. of AAAI-97, 372-377.

Cha, B., and Iwama, K. 1995. Performance Test of Local Search Algorithms Using New Types of Random CNF Formulas. In Proceedings of IJCAI-95, 304-310.

Cha, B., and Iwama, K. 1996. Adding New Clauses for Faster Local Search. In Proc. of AAAI-96, 332-337.

Frank, J. 1996. Weighting for Godot. In Proceedings of AAAI-96, 338-343.

Frank, J. 1997. Learning Short-Term Weights for GSAT. In Proceedings of AAAI-97, 384-389.

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

Morris, P. 1993. The Breakout Method for Escaping from Local Minima. In Proceedings of AAAI-93, 40-45.

Selman, B., and Kautz, H. 1993. Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In Proceedings of IJCAI-93, 290-295.

Selman, B.; Levesque, H.; and Mitchell, D. 1992. A New Method for Solving Hard Satisfiability Problems. In Proceedings of AAAI-92, 440-446.

Thornton, J., and Sattar, A. 1997. Applied Partial Constraint Satisfaction using Weighted Iterative Repair. In Sattar, A. (ed.) *Advanced Topics in Artificial Intelligence*, LNAI 1342, 57-66: Springer-Verlag.