

The Backbone of the Travelling Salesperson

Philip Kilby
ANU
Canberra, Australia
Philip.Kilby@anu.edu.au

John Slaney
NICTA and ANU
Canberra, Australia
John.Slaney@anu.edu.au

Toby Walsh
NICTA and UNSW
Sydney, Australia
tw@cse.unsw.edu.au

Abstract

We study the backbone of the travelling salesperson optimization problem. We prove that it is intractable to approximate the backbone with any performance guarantee, assuming that $P \neq NP$ and there is a limit on the number of edges falsely returned. Nevertheless, in practice, it appears that much of the backbone is present in close to optimal solutions. We can therefore often find much of the backbone using approximation methods based on good heuristics. We demonstrate that such backbone information can be used to guide the search for an optimal solution. However, the variance in runtimes when using a backbone guided heuristic is large. This suggests that we may need to combine such heuristics with randomization and restarts. In addition, though backbone guided heuristics are useful for *finding* optimal solutions, they are less help in *proving* optimality.

1 Introduction

In recent years, there has been much research into what makes a search problem hard. In decision problems, we now have a rich picture based upon rapid transitions in solubility and a corresponding thrashing in search algorithms for the critical constrained problems close to such “phase transitions” [Cheeseman *et al.*, 1991; Mitchell *et al.*, 1992]. The picture is much less clear for optimization. Optimization problems do not have a transition in solubility, since optimal solutions always exist. A strong candidate to replace the transition in solubility is a transition in the backbone size [Monasson *et al.*, 1998]. In this paper, we study the complexity of computing and of approximating the backbone. We also look at using such backbone information to help find optimal solutions, and to prove optimality. Throughout this paper, we focus on the symmetric travelling salesperson (TSP) optimization problem. This is the problem of computing the shortest tour which visits every city, where the inter-city distance matrix is symmetric. However, most of our results easily generalise to the asymmetric case.

2 Backbones

In decision problems, the concept of the *backbone* has proven to be very useful in understanding problem hardness. For example, the backbone of a satisfiability (SAT) problem is the set of literals which are true in every model [Monasson *et al.*, 1998]. Backbone size appears closely correlated to problem hardness [Parkes, 1997; Monasson *et al.*, 1998]. If a SAT problem has a large backbone, there are many opportunities to assign variables incorrectly. Such problems tend to be hard therefore for systematic methods like Davis-Putnam. A large backbone also means that solutions are clustered. Such problems therefore can be hard to solve with local search methods like WalkSAT.

Backbones have been less well studied in the context of optimization. In [Slaney and Walsh, 2001], the backbone of an optimization problem is defined to be the *frozen decisions*: those with fixed outcomes for all optimal solutions. For example, the backbone of a TSP problem is the set of edges which occur in all tours of minimal cost. Such a concept again seems useful in understanding problem hardness. For instance, the cost of finding optimal (or near optimal) solutions is positively correlated with backbone size [Slaney and Walsh, 2001].

3 Computing the Backbone

It is not difficult to see that it is NP-hard to find the backbone of a TSP problem. If the optimal tour is unique, then the backbone is complete. A procedure to compute the backbone then trivially gives the optimal tour length. Complications arise when the optimal tour is not unique and the backbone is incomplete. For example, suppose there are two disjoint but optimal tours. The backbone is then empty. Nevertheless, we can find the optimal tour length using a polynomial number of calls to a procedure that determines backbone edges.

Theorem 1 *The TSP BACKBONE problem (the problem of deciding if an edge is in the backbone of a TSP problem) is NP-hard.*

Proof: We rescale the TSP problem so that only one optimal tour remains. Without loss of generality, we assume that inter-city distances are integers. First, for a problem with n cities, we multiply each inter-city distance by $2^{n(n+1)}$. This gives $n(n+1)$ new bits in the least significant digits of each

number. Note that this only requires polynomial space. We divide these new bits into n sections, one for each city, of length $n + 1$ bits. For the inter-city distance between city i and city j , we set the j th bit in the i th section and the i th bit in the j th section. Each tour length now encodes the list of edges visited. In particular, the i th section of the tour length has two bits, say, j and k set. This represents the fact that the tour includes edges from j to i and then to k . Even if the old distance matrix had several optimal tours, the new distance matrix only has one of them. Let c_i be the two cities on a tour connected to the i th city. Then the optimal and unique tour of the rescaled problem is the optimal tour of the unrescaled problem in which $\langle \max(c_1), \min(c_1), \dots, \max(c_n), \min(c_n) \rangle$ is lexicographically least. As the optimal tour is now unique, the backbone is complete. We can compute this by a polynomial number of calls to a procedure for deciding if an edge is in the backbone. $\square$

It remains open whether the TSP BACKBONE problem is NP-complete or not. Completeness would seem to require a short witness that a tour was optimal. We can, however, say it is both NP-hard and NP-easy. It is NP-easy as deciding if an edge is in the backbone can be reduced to a polynomial number of calls to a TSP decision procedure. Garey and Johnson suggest that problems which are both NP-hard and NP-easy might be called NP-equivalent [Garey and Johnson, 1979]. Although this class contains problems which do not belong to NP, the class has the property of NP-complete decision problems that: unless $P=NP$, no problem in the class can be solved in polynomial time, and if $P=NP$ then all problems in the class can be solved in polynomial time. In other words, the TSP BACKBONE problem is polynomial if and only if $P=NP$.

4 Approximating the Backbone

Even though computing the backbone is intractable in general, might we be able to approximate it?

4.1 Sound approximation

Suppose that we have an approximation procedure that returns some subset of the backbone edges. It is easy to see that, assuming $P \neq NP$, no such procedure can be guaranteed to return a fixed fraction of the backbone in polynomial time: First, we rescale the distance matrix as before so that the optimal tour is unique and the backbone complete. The sound approximation procedure could be called to return at least one backbone edge - say (a, b) . Create a new TSP by collapsing a and b into a single node a' . The cost $c(a', x)$ for node x would be set to $\min(c(a, x), c(b, x))$; all other costs as before. The procedure could be repeatedly called to construct the optimal tour edge by edge in polynomial time.

A similar argument shows that no sound approximation procedure can be guaranteed to return at least one backbone edge if the backbone is non-empty in polynomial time.

4.2 Unsound approximation

Suppose that edges returned by an approximation procedure are not guaranteed to be in the backbone. If we do not limit the number of edges incorrectly returned, then there exists a polynomial time approximation that meets any approximation ratio (that is, returns any given fraction of the backbone).

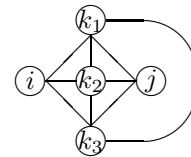
For example, the approximation procedure that returns all $O(n^2)$ possible edges returns all the backbone. We therefore consider approximation procedures which limit the number of edges falsely assigned to the backbone. An approximation procedure is a “majority-approximation” iff, when the backbone is non-empty, more edges are returned that come from the backbone than do not come from the backbone. If the backbone is empty, any number of edges can be falsely returned.

Theorem 2 *If $P \neq NP$ then no majority-approximation procedure can be guaranteed to return a fixed fraction α or greater of the backbone edges in polynomial time.*

Proof: We show how such a procedure could determine if a graph (V, E) has a Hamiltonian path with a designated starting and ending vertex in polynomial time, contradicting $P \neq NP$. We construct a TSP problem which can have two sorts of shortest tours. For every Hamiltonian path between the starting and ending vertex, there is a corresponding tour of length $n - 1$ where $|V| = n$. These tours have a non-empty set of backbone edges taken from some set S . On the other hand, if there is no Hamiltonian path between the starting and ending vertex, the shortest tour is of length n , and has a non-empty set of backbone edges (denoted T) disjoint with S .

Since the approximation procedure returns at least a fixed fraction of the backbone, it returns at least one correct backbone edge. As it is a majority-approximation procedure, the number of edges in the backbone correctly returned is more than the number incorrectly returned. By computing the ratio of the number of edges of the two types returned, we can determine if the backbone is drawn from edges in S or from those in T . That is, we can determine if there is a Hamiltonian path or not in polynomial time.

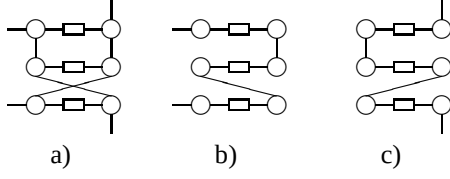
We need two special gadgets. The first is a “backbone free” connector. This connects together two nodes without introducing any backbone edges. For example, to connect node i to node j , we use the following small circuit which introduces 3 new intermediate nodes: k_1 , k_2 and k_3 . All marked edges are of cost 0, and all unmarked edges are of cost n^2 :



A tour from i to j goes through some permutation of k_1 , k_2 , k_3 . No edge is in all possible tours, and thus no edge appears in the backbone. Note that we can modify this (and the other circuits) to have non-zero edges costs by increasing all other edge costs appropriately. We will draw this gadget as a rectangular box between nodes i and j .

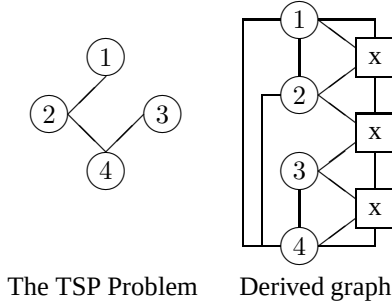
The second gadget is a “switch” circuit. This uses three of the backbone free connector gadgets. Four edges go into this circuit, two horizontally and two vertically. The switch has two modes. In the first mode, the tour enters and exits by the vertical edges. In the other mode, the tour enters and exits by the horizontal edges. The switch circuit contains 15 nodes, all of which are visited in both modes. Note that no backbone edges are common between the two modes. The

gadget is shown below in a). As before, all marked edges are of cost 0, and all unmarked edges are of cost n^2 . We give the two different modes of the circuit in b) and c). We will draw this switch gadget as a square box containing an “x”.



We use these gadgets to construct a TSP problem with the required shortest tours. The problem has $16(n-1)+1$ nodes, n of which correspond to vertices in the graph (V, E) and the rest are in $n-1$ switch gadgets. If the graph (V, E) has an edge between vertex i and j then the TSP problem has an edge of cost 1 between node i and j . We shall assume that the starting and ending vertices/nodes are 1 and n respectively. To finish the tour, we put a zero cost edge between node 1 and n . There are also $n-1$ switch circuits. The horizontal wires in the i th switch circuit are connected to node i and $i+1$ ($1 \leq i \leq n-1$). The vertical wires in the j th switch circuit are connected to the $j-1$ th and $j+1$ th switch circuit ($1 < j < n-1$). The vertical wires in the 1st switch circuit are connected to node 1 and the 2nd switch circuit. The vertical wires in the $n-1$ th switch circuit are connected to the $n-2$ th switch circuit and to node n . All edge costs between switch circuits and nodes, and between switch circuits are zero except for the edge between node 1 and the horizontal input to the 1st switch circuit which has cost n . The cost between any two unmarked nodes is n^2 as before.

We give an example for $n=4$ in which the graph (V, E) has edges between nodes 1 and 2, 3 and 4, and 2 and 4:



If there is a Hamiltonian path in the graph, we follow the corresponding path in the TSP starting with node 1 and ending at node n . We then exit from node n , and enter the vertical wire in the $n-1$ th switch, and continue through the vertical wires to switch 1 where we exit and take the zero cost edge back to node 1. This completes a tour of length $n-1$. On the other hand, if there is not a Hamiltonian path, the shortest tour visits nodes 1 to n by alternating with the $n-1$ switch circuits. It then takes the zero cost edge from node n back to node 1. This completes a tour of length n . $\square$

Similar arguments show that if $P \neq NP$ then no majority-approximation procedure can be guaranteed to return at least one backbone edge when the backbone is non-empty in polynomial time

5 Epsilon Backbone

Despite these negative complexity results, backbones may still be easy to compute in practice. We study here the possibility of using approximation procedures. We define the ϵ -backbone of a TSP problem as the set of edges which occur in all tours within a factor $(1+\epsilon)$ of the optimal. In Figure 1, we plot the fractional size of the ϵ -backbone (that is, the size of the ϵ -backbone normalized by n) against $1+\epsilon$ for random Euclidean TSP problems.

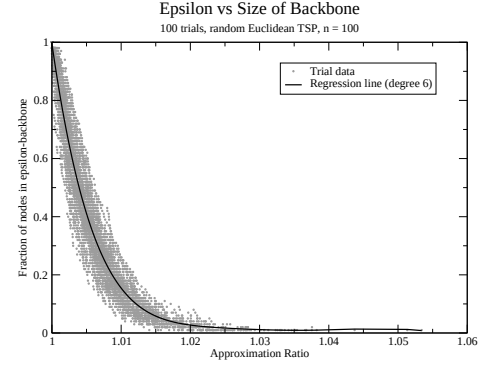


Figure 1: Fractional size of the ϵ -backbone (y-axis) plotted against approximation ratio ($= 1 + \epsilon$) for 100 random Euclidean TSP problems with 100 nodes. Degree 6 regression line fitted.

We see that tours within 5% of optimal have approximately 40% of the backbone of optimal tours. We observe similar results with non-random instances taken from TSPLib. It appears that much of the backbone of a TSP problem is present when we are near to the optimal solution. As heuristics can often find near optimal solutions in a short time, it may be easy to compute a large part of the backbone in practice.

6 Approximation Methods

We now see if TSP heuristics can be used as the basis of an approximation method for computing the backbone. To compute if an edge is in the backbone or not, we can commit to the edge and compute the optimal tour, and then throw the edge out and re-compute the optimal tour. The edge is in the backbone iff the first tour is shorter in length than the second. Suppose now that instead of computing the two optimal tour lengths, we run some good heuristic method like [Lin and Kernighan, 1973] for a polynomially bounded time. If it is true that the heuristic is good, it will frequently find close to optimal length tours (though we have no guarantee that it does). This yields a simple approximation method for computing edges that are likely to be in the backbone.

Tests were run on random Euclidean TSP problems of size 100, 250 and 500 nodes. Most problems have unique solutions, and hence complete backbones. Exceptions were: 6 problems of size 100, 16 of size 250 and 19 of size 500 had backbones of size $< n$. Table 1 shows the percentage of the backbone correctly identified using the heuristic procedure, as well as the percentage of “false positive” results. The Lin-Kernighan heuristic is very accurate at these problem sizes, so

		n = 100	n = 250	n = 500
Ave Approx Ratio		1.0000	1.0002	1.0008
Correct	10%	99.0	69.4	28.8
(%)	Median	100.0	100.0	75.8
	90%	100.0	100.0	100.0
False	10%	0.0	0.0	0.0
Positive	Median	0.0	0.0	0.0
(%)	90%	1.0	0.4	0.6

Table 1: Estimation of backbone using the Lin-Kernighan heuristic. 100 trials of random Euclidean TSP problems at each size. Approx Ratio is mean (heuristic-solution)/(optimal-solution) of original problem.

approximation ratios were small. The heuristic is very good at identifying backbone edges and gives few false-positives. Its main weakness is missing backbone edges.

7 Backbone Guided Heuristics

One motivation for identifying backbone is to try to reduce search. For example, Climer and Zhang use backbone variables in asymmetric TSP problems to preprocess and simplify [Climer and Zhang, 2002]. As a second example, Dubois and Dequen use a backbone guided heuristic to solve 700 variable hard random 3 SAT problems [Dubois and Dequen, 2001]. In the following section, we show that whilst backbone guided heuristics can be helpful in finding optimal solutions, we must use them with care. Median runtime are good, but there are a few long runs (suggesting that a randomization and restarts strategy may be useful). In addition, we show both experimentally and theoretically, that backbone guided heuristics can be very poor when it comes to proving optimality. To make the demonstration very direct, we use a method for solving TSP problems that relies heavily on the branching heuristic. The method does not use the “cuts” that characterise state-of-the-art solvers. However, it lets us compare directly the efficacy of various branching decisions.

We use a branch and bound solver with lower bounds provided by Lagrangean relation with 1-trees [Reinelt, 1994]. Upper bounding is by a single, deterministic run of Or-opt [Or, 1976] testing all forward and reverse moves of blocks of size $n/2$ down to 1. The MST bound is enhanced by eliminating all other edges into a node if two incident edges are already forced into the solution. Depth-first search is used, with the node having the least lower bound explored. The algorithm branches on an edge currently in the tour representing the upper bound. The edge can be chosen in one of several ways:

bb We exclude in turn every possible edge from the solution. A comparison is then made between the new objective and the current upper bound. If the new objective is reduced, then an improvement to the upper bound has been found. If such an improving move is found, the edge giving the largest improvement is chosen. If no such improvement is found, then the edge giving the largest *increase* in objective is most likely to be in the backbone, so that edge is chosen.

freqbb A frequency table is kept during search. As each new upper-bound tour is created, the frequency of appearance

Method	Mean		Median	
	Times/s	Nodes	Time/s	Nodes
bb	60.9	685.8	3.0	5.0
freqbb	98.5	1618.1	5.0	174.0
long	34.0	436.8	0.0	4.0
longish	21.8	332.0	0.0	5.0
next	38.0	642.8	1.0	19.0
rand	58.1	980.5	2.0	40.0
short	61.1	909.8	7.0	113.0

Table 2: Branching heuristics – finding the optimal solution

Method	Mean		Median		limit exceeded
	Time/s	Nodes	Time/s	Nodes	
bb	199.1	2559.8	35.0	244.0	12
freqbb	206.5	3284.9	28.5	422.0	14
long	360.3	4816.7	51.6	755.0	26
longish	87.7	1946.1	12.5	130.0	4
next	93.7	1775.3	10.1	108.0	4
rand	179.1	2859.3	24.4	348.0	10
short	199.8	2921.9	35.8	413.0	11

Table 3: Branching heuristics – proving optimality

of each edge is updated. Edges with higher frequency are more likely to be backbone edges, so the branch choice is the edge with the greatest frequency.

next The next unused edge in the tour is chosen

long The longest unused edge in the tour is chosen

short The shortest unused edge in the tour is chosen

longish A method suggested by the results of this test. Use *long* for the first n branches, then use *next*.

rand A random edge is chosen. As this is non-deterministic, the heuristic was run five times on each problem.

A branch-and-bound solver has two phases: finding the optimal solution, and proving optimality. Different heuristics do better in different phases, so we report the results separately in Table 2, and Table 3. We used 50 node random Euclidean TSP problems with a time limit of 1000 seconds on each run. Table 3 shows the number of times this limit was exceeded. The time and node results *include* problems for which the time limit was exceeded.

These results show a “heavy-tailed” distribution [Gomes *et al.*, 2000] with many problems being solved quickly (hence a low median) but a few taking a very long time (hence a high average). Randomised algorithms and restarts have been shown to reduce the average time in NP-hard problems [Meisels and Kaplansky, 2004; Selman *et al.*, 1994]. However, we wished to look at the “pure” algorithms for comparison.

These results show that the bb heuristic is quite effective at finding the optimal in terms of the median number of nodes visited. However, the average is quite large, indicating the method could benefit from multiple restarts. This reduced number of nodes comes at a price in terms of runtime. Other methods, such as *long* outperform it in runtime. In terms

of proving optimality, the *bb* heuristic performed relatively poorly, being little better than random.

8 Pathological Example

As a further caution to using backbone guided branching heuristics, we present a pathological instance of the TSP problem. If we branch on a non-backbone edge, this instance can be solved in a *single* branch. However, if we branch first on backbone edges, we visit an *exponential* number of branches before proving optimality. Since a search space is defined by a particular algorithm, we must define a solution method. The following algorithm is sensible, but simple enough that we can predict its behavior theoretically. It lacks the refined LP-based cuts that allow modern TSP codes to solve huge problems, but is a fairly standard basic algorithm ([Reinelt, 1994]). It has the following features:

1. Uses branch and bound.
2. Stops when the lower bound equals the upper bound.
3. Upper bound provided by current best tour.
4. Lower bound provided by 1-tree relaxation. A 1-tree is simply a minimum spanning tree (MST) with one extra edge added from a leaf to another node already in the tree. The 1-tree contains exactly one cycle, which may be a complete tour.
5. If two edges are forced to be incident to a single node, then the lower-bounding procedure does not consider any other edges incident to that node.
6. Preprocessing identifies nodes which have only two “useful” incident edges. These two are forced into the solution. A “useful” edge here is defined to be an edge with cost less than any available upper bound (an edge with cost greater than an estimate for the entire tour will never be used in an optimal solution). An initial upper bound can be calculated using the tour (1, 2, 3, ...).

The TSP problem is given in Figure 2(a). Node 6 can be “cloned” arbitrarily many times, and has been cloned at nodes 7 and 8. The clones connect to each neighbour at cost 20, node 1 at cost 10, and all other clones at cost 21. Arcs not shown have cost $1000 + 20n$, and hence will never be used in an optimal solution. The upper bound for this graph is given by the tour 1, 2, .. 10, 1 with cost 200. The minimum spanning tree is shown in Figure 2(b) and has cost 130. The bold edges (10,1) and (10,9) are fixed in due to the preprocessing. The one-tree adds, for example (3,4) at cost 20. The lower bound is therefore 150.

Forcing the non-backbone edge (1,2) excludes the “spoke” edges at nodes 1, as node 1 now has 2 edges incident. The MST (shown in Figure 3(a)) now has cost 180, and the 1-tree bound (using edge (8,9) or (3,4) to connect) is 200. As the lower bound is the same as the upper bound, the branch and bound algorithm completes. So if (1,2) (or, by a symmetric argument (1,3)) were chosen as the branch node, the procedure terminates immediately. On the other hand, branching on backbone edges would force in edges like (5,6). If 6 were cloned as described above, there could be arbitrarily many of these edges. Figure 3(b) shows the effect on the lower bound

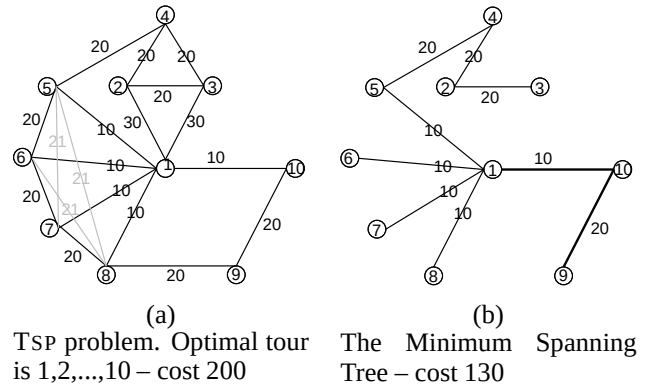


Figure 2: The TSP problem, and its MST

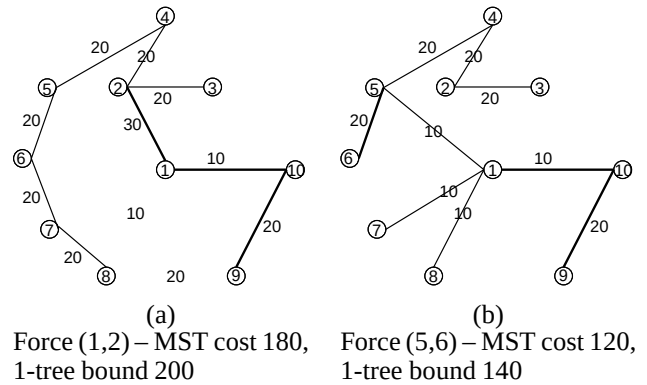


Figure 3: Forcing edges. Forced edges in bold.

of forcing edge (5,6) into the solution. Forcing (5,6) gives an MST with cost 140. The 1-tree is connected using an edge like (6,7) or (3,4) with cost 20, giving a lower bound of 160. If there were other clones of 6, when they were forced into the solution, then the lower bound would not approach the upper bound until edges between *all* clones have been forced. Forcing out (5,6) leaves a problem that looks very similar to the original. It still has backbone edges round the perimeter, but an MST/1-tree which uses the “spoke”-type edges around node 1 and hence gives an unachievable lower bound.

On backtracking, forcing out (5,6) gives the same lower bound as the parent problem. A new upper bound can be found using the cost 21 edges. As the lower-bound is still less than the upper, we can not stop. A recursive argument shows that the right branch must be completely explored before it can be excluded. It is not until all backbone edges are forced in, or forced out, and the algorithm moves on to branch on edge (1,2) or (1,3) that the optimal is determined. Hence the backbone heuristic would need to visit an exponential number of branches to solve the problem.

9 Related Work

Slaney and Walsh demonstrated that the cost of finding optimal (or near optimal) solutions is positively correlated with backbone size [Slaney and Walsh, 2001]. However, they also showed that the cost of proving optimality is negatively cor-

related with backbone size. If we have a small backbone, then there are many optimal and near-optimal tours. An algorithm like branch and bound has to do a lot of work to ensure there are no shorter tours.

Zhang has shown that asymmetric random TSP problems undergo a phase transition in the tour cost and backbone size as the precision of the distance matrix is varied [Zhang, 2004]. He argues that similar results hold for the symmetric TSP problem, and for structured TSP problems provided distances are drawn from a common distribution. The search cost of his branch-and-bound solver also changes from easy to hard as the precision of the distance matrix is increased and the backbone size increases.

Climer and Zhang used an approximation method they call “limit-crossing” to identify backbone variables in asymmetric TSP problems (as well as their dual, “fat” variables which are not part of any optimal solution) [Climer and Zhang, 2002]. By committing to backbone variables and eliminating fat, we can reduce the size of the problem and thereby reduce search.

Beacham has considered the complexity of computing the backbone for a range of decision problems like the satisfiability and Hamiltonian path problem [Beacham, 2000]. He considers a slightly modified definition of backbone: the set of decisions whose negation give an unsatisfiable subproblem. This definition is equivalent to the usual one for satisfiable problems only. He shows that recognising when the backbone is empty is NP-complete.

Zhang has demonstrated experimentally that there is a sharp transition in the size of the backbone of random MAX 3SAT problems [Zhang, 2001]. This appears to be correlated with the transition in the random 3SAT decision problem.

10 Conclusion

We have looked at the backbone of the travelling salesperson problem. It is not hard to see that computing the backbone is NP-hard. However, we have also shown that it is hard to approximate the backbone with any performance guarantee (assuming $P \neq NP$ and there is a limit on the number of edges falsely returned). Nevertheless, in practice, it appears that much of the backbone is present in close to optimal solutions. Approximation methods based on good heuristics can often therefore find much of the backbone. Such backbone information can be used to guide the search for an optimal solution. However, it should be used with care as the variance in runtimes when using a backbone guided heuristic is large. In addition, backbone guided heuristics are less good at proving optimality.

What general lessons can be taken from this study? First, these intractability results are likely to generalize to other problems domains. We conjecture that it will also be hard to compute or approximate the backbone in other optimization problems (both fundamental theoretical problems like MAX-SAT, and more practical problems like job shop scheduling). Second, we predict that much of the backbone will also be present in close to optimal solutions in these other domains. Approximation based methods may also work well in these cases. Third, whilst backbones are a useful concept in explaining some aspects of problem hardness in optimization, it

is clear that other concepts are still needed. A number of candidate measures have been proposed for decision problems (e.g. backdoor variables) that have yet to be explored for optimization.

References

- [Gomes *et al.*, 2000] Carla Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.
- [Lin and Kernighan, 1973] S. Lin and B. Kernighan. An effective heuristic for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.
- [Meisels and Kaplansky, 2004] Amnon Meisels and Eliezer Kaplansky. Iterative restart technique for solving timetabling problems. *European Journal of Operational Research*, 153(1):41–50, 2004.
- [Or, 1976] I. Or. *Travelling Salesman-Type Combinatorial Problems and Their Relation to the Logistics of Blood-Banking*. PhD thesis, Dept of Industrial Eng. and Management Sciences, Northwestern University, Evanston, IL., 1976.
- [Reinelt, 1994] Gerhard Reinelt. *The Travelling Salesman. Computational Solutions for TSP Applications, Lecture Notes in Computer Science* vol. 840 Springer-Verlag, 1994.
- [Beacham, 2000] A.J. Beacham. The complexity of problems without backbones. Master’s thesis, Dept. of Computing Science, University of Alberta, 2000.
- [Cheeseman *et al.*, 1991] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proc. of 12th IJCAI*, pages 331–337. 1991.
- [Climer and Zhang, 2002] S. Climer and W. Zhang. Searching for backbones and fat: A limit-crossing approach with applications. In *Proc. of 18th National Conf. on AI*. 2002.
- [Dubois and Dequen, 2001] O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hardn 3-SAT formulae. In *Proc. of 17th IJCAI*, pages 248–253. 2001.
- [Garey and Johnson, 1979] M.R. Garey and D.S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness*. W.H. Freeman, 1979.
- [Mitchell *et al.*, 1992] D. Mitchell, B. Selman, and H. Levesque. Hard and Easy Distributions of SAT Problems. In *Proc. of 10th National Conf. on AI*, pages 459–465. 1992.
- [Monasson *et al.*, 1998] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity for characteristic ‘phase transitions’. *Nature*, 400:133–137, 1998.
- [Parkes, 1997] A. Parkes. Clustering at the phase transition. In *Proc. of the 14th National Conf. on AI*, pages 340–345. 1997.
- [Selman *et al.*, 1994] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of AAAI’94*, pages 337–343, 1994.
- [Slaney and Walsh, 2001] J. Slaney and T. Walsh. Backbones in optimization and approximation. In *Proceedings of 17th IJCAI*. 2001.
- [Zhang, 2001] W. Zhang. Phase transitions and backbones of 3-SAT and Maximum 3-SAT. In *Proc. of 7th Int. Conf. on Principles and Practice of Constraint Programming (CP2001)*. Springer, 2001.
- [Zhang, 2004] W. Zhang. Phase transitions and backbones of the asymmetric traveling salesman problem. *Journal of Artificial Intelligence Research*, 21:471–497, 2004.

Complete MCS-Based Search: Application to Resource Constrained Project Scheduling

Philippe Laborie

ILOG

9, rue de Verdun, 94253 Gentilly Cedex, France

plaborie@ilog.fr

Abstract

This paper describes a simple complete search for cumulative scheduling based on the detection and resolution of *minimal critical sets* (MCS). The heuristic for selecting MCSs relies on an estimation of the related reduction of the search space. An extension of the search procedure using *self-adapting shaving* is proposed. The approach was implemented on top of classical constraint propagation algorithms and tested on resource constrained project scheduling problems (RCPSP). We were able to close more than 15% of the previously open problems of the PSPLIB [Kolisch and Sprecher, 1996] and improve more than 31% of the best known lower bounds on those heavily studied problems. Other new results on open-shop and cumulative job-shop scheduling are reported.

1 Introduction

The resource constrained project scheduling problem (RCPSP) is one of the most general scheduling problem that is extensively studied in the literature. It consists in scheduling a project, which is a set of activities linked with precedence constraints, by means of a set of limited resources while minimizing the total duration of the project. The decision variant of the RCPSP, i.e., the problem of determining whether there exists a feasible project of makespan smaller than a given deadline, is NP-hard in the strong sense. The RCPSP is a very popular and frequently studied NP-hard optimization problem and the last 20 years have witnessed a tremendous improvement of both heuristic and exact solution procedures (cf. e.g. the recent surveys given in [Demeulemeester and Herroelen, 2002; Hartmann and Kolisch, 2005]). The currently best lower bounds on the makespan for the general RCPSP are based on solving linear programs using adequate cutting planes [Brucker and Knust, 2000; Baptiste and Demassey, 2004]. State-of-the-art techniques for upper-bounds rely on meta-heuristics such as Genetic Algorithms, Ant Colony Optimization or Large Neighborhood Search. Many scheduling problems such as job shop, cumulative job shop and open-shop can be modeled as special cases of RCPSP.

In this article, we present a pure constraint programming approach based on the exploration of a complete search tree to prove that the project cannot be achieved within a given deadline or to exhibit a feasible project if one exists. The search procedure is based on the detection and resolution of *minimal critical sets* (MCS) [Laborie and Ghallab, 1995] at each node of the search. MCSs are carefully chosen using a heuristic that tries to minimize the size of the search space. During the search, strong constraint propagation is enforced using classical scheduling constraint propagation techniques such as *time-tabling*, *edge-finding*, *precedence energy* and *balance* constraints [Laborie, 2003].

Next section recap the definition of the resource constrained project scheduling problem and introduces some notations. Section 3 describes our basic search procedure as well as the heuristic to select MCSs. Section 4 extends the basic search procedure to perform *self-adapting shaving*. The last part of the paper consists of experimental results on classical benchmarks (general RCPSP, open-shop and cumulative jobshop problems). For general RCPSP, we show that our approach closes more than 15% of previously open instances and improves more than 31% of best known lower bounds of the famous PSPLIB instances [Kolisch and Sprecher, 1996]. The same approach using exactly the same settings was used to close all the hard open-shop instances of [Gu  ret and Prins, 1999] in less than 5s CPU time and to improve the best known lower bounds and close several instances of cumulative jobshop [Nuijten, 1996].

2 Model and notations

The resource constrained project scheduling problem (RCPSP) can be formally stated as follows. A project is made of a set of activities $\mathcal{A}$ linked by precedence constraints. Precedence constraints can be represented by a directed acyclic graph $G = (\mathcal{A}, \mathcal{E})$ where each node in $\mathcal{A}$ represents an activity and each arc $(A, B) \in \mathcal{E}$ represents a precedence constraint between A and B . Let $d(A)$ denote the fixed duration of activity $A \in \mathcal{A}$ and $s(A)$ (resp. $e(A)$) denote the decision variable representing the start (resp. end) time of activity A . A set of discrete capacity resources $\mathcal{R}$ is considered, each resource $R \in \mathcal{R}$ having a maximal available capacity $Q(R)$ over the entire scheduling horizon. Each activity $A \in \mathcal{A}$ requires a non-negative quantity $q(A, R)$ of resource R . The problem is to find a feasible instantiation s

of the activity start times such that precedence and resource constraints are satisfied and the schedule makespan is minimal. More formally:

$$\begin{aligned}
& \text{minimize} && \max_{A \in \mathcal{A}} e(A) \\
& \text{subject to :} && \\
& \forall A \in \mathcal{A}, && 0 \leq s(A) \\
& && e(A) = s(A) + d(A) \\
& \forall (A, B) \in \mathcal{E}, && e(A) \leq s(B) \\
& \forall R \in \mathcal{R}, \forall t \in \mathbb{Z}^+, && \sum_{A \in \mathcal{S}(t)} q(A, R) \leq Q(R)
\end{aligned}$$

where $\mathcal{S}(t)$ is the set of activities executing at time t :

$$\mathcal{S}(t) = \{A \in \mathcal{A}, s(A) \leq t < e(A)\}$$

A resource requirement of activity A on resource R is a triple $u = (A, R, q)$ where $q = q(A, R) > 0$. If $u = (A, R, q)$ is a resource requirement, we will denote $A(u) = A$ the activity of u , $R(u) = R$ the required resource, $q(u) = q$ the required quantity, $s(u)$ (resp. $e(u)$) will denote the start (resp. end) time of the activity of u . We will also denote $U(R) = \{u/R(u) = R\}$ the set of resource requirements on resource R . If $\varphi \subseteq U(R)$ is a subset of resource requirements on a resource R , we will denote $q(\varphi) = \sum_{u \in \varphi} q(u)$ the global resource consumption of R by activities in φ .

3 Search

3.1 Branching scheme

Our branching scheme assumes that a temporal network representing the relations between the time-points of all activities (start and end) using the point algebra of [Vilain and Kautz, 1986] is maintained during the search. In our implementation, this is ensured by the precedence graph constraint of ILOG SCHEDULER [ILOG, 2005]. We denote $\{\emptyset, \prec, \preceq, =, \succ, \succeq, \neq, ?\}$ the set of qualitative relations between time points. If u and v represent two resource requirements, we will denote $u \preceq v$ if and only if $e(u) \leq s(v)$.

The branching scheme relies on the notion of minimal critical sets (MCS) and their resolvers as introduced in [Laborie and Ghallab, 1995]. A MCS is a minimal set of resource requirements on the same resource R that could be executed simultaneously and would, in this case, over-consume resource R . MCSs are a natural generalization to cumulative scheduling of the pairs of activities conflicting for the same unary resource in disjunctive scheduling.

Definition 1 (Minimal critical set) A minimal critical set on a resource R is a subset $\phi \subseteq U(R)$, such that:

1. $Q(R) < q(\phi)$
2. $\forall \varphi \subsetneq \phi, q(\varphi) \leq Q(R)$
3. $\bigwedge_{(u,v) \in \phi \times \phi} s(u) \prec e(v)$ is consistent with the current temporal network

Informally, the different ways to resolve a minimal critical set consist in posting a precedence constraint between any two of its activities.

Definition 2 (Resolvers of a minimal critical set) If $\phi \subseteq U(R)$ is a MCS, we call resolvers of ϕ the set of temporal constraints $Res(\phi) = \{u \preceq v : (u, v) \in \phi^2, u \neq v\}$.

As described in [Laborie and Ghallab, 1995], the set of resolvers $Res(\phi)$ of a MCS ϕ can be simplified so as to remove those resolvers $\rho \in Res(\phi)$ for which there exists another resolver $\rho' \in Res(\phi)$ such that $\rho \Rightarrow \rho'$ given the current temporal network. Indeed, in such case, the resolver ρ is redundant. Such a simplification procedure can be achieved in $O(k^3)$ if k is the size of the MCS using the naive Algorithm 1. Line 7 allows removing resolver $u \preceq v$ if there exists w such that $u \preceq v \Rightarrow u \preceq w$ or $u \preceq v \Rightarrow w \preceq v$. In what follows, we assume that the set of resolvers of a MCS has been simplified.

Algorithm 1 Resolver simplification algorithm

```

1: procedure SIMPLIFY_RESOLVERS( $\phi$ )
2:    $Res(\phi) \leftarrow \emptyset$ 
3:   for all  $u$  in  $\phi$  do
4:     for all  $v$  in  $\phi \setminus \{u\}$  do
5:        $keep_{uv} \leftarrow \text{TRUE}$ 
6:       for all  $w$  in  $\phi \setminus \{u, v\}$  do
7:         if  $s(v) \preceq s(w)$  or  $e(w) \preceq e(u)$  then
8:            $keep_{uv} \leftarrow \text{FALSE}$ 
9:         break
10:      if  $keep_{uv}$  then
11:         $Res(\phi) \leftarrow Res(\phi) \cup (u \preceq v)$ 
12:   return  $Res(\phi)$ 

```

At each search node, our branching scheme consists in selecting a MCS ϕ and branching on its possible resolvers in the children nodes until there is no more MCS. This approach is clearly complete.

3.2 Heuristics

As all the resolvers consist of temporal constraints of the form $x \preceq y$ where x and y are two time-points (start or end of an activity), we are interested in an estimation of the size of the search space after posting such a precedence constraint. The fraction of the search space that is preserved when adding a precedence constraint is estimated using the complementary of the commitment measure introduced in [Laborie, 2003].

Let x and y be two time-points with respective lower and upper bound for time value: $[x_{min}, x_{max}]$ and $[y_{min}, y_{max}]$. The size of the search space is estimated by the Cartesian product of the domain of the two variables, that is, the area of the rectangle $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. The size of the search space that is preserved when adding the constraint $x \preceq y$ is the part of that rectangle above the line $x = y$ as illustrated in Figure 1. The fraction of the search space that is preserved can thus be estimated as follows. Let:

$$A = (y_{max} - y_{min} + 1)(x_{max} - x_{min} + 1)$$

$$B = (y_{max} - x_{min} + 1)(y_{max} - x_{min} + 2)$$

$$C_{min} = \max(0, (y_{min} - x_{min})(y_{min} - x_{min} + 1))$$

$$C_{max} = \max(0, (y_{max} - x_{max})(y_{max} - x_{max} + 1))$$

The fraction is then equal to:

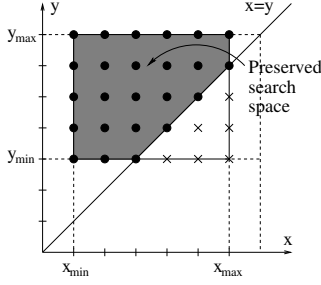


Figure 1: Preserved search space when adding $x \preceq y$

$$preserved(x \preceq y) = \frac{B - C_{min} - C_{max}}{2 \cdot A}$$

In the example of Figure 1, this gives $A = 30$, $B = 56$, $C_{min} = 6$, $C_{max} = 2$, and $preserved(x \preceq y) = 24/30$.

If ω is the size of the search space below the current search node, the size of the search space after posting a temporal constraint $x \preceq y$ can be estimated by $\omega \cdot preserved(x \preceq y)$. If ϕ is the MCS that is selected to be resolved at the current search node, the size of the search space to explore below the current node can thus be estimated as the sum of the sizes of the search space below each child node, that is: $\omega \cdot \sum_{\rho \in Res(\phi)} preserved(\rho)$. Therefore, $preserved(\phi) = \sum_{\rho \in Res(\phi)} preserved(\rho)$ estimates the fraction of the search space that is preserved when choosing ϕ as the next MCS to solve¹. Our heuristic simply chooses to resolve next the MCS ϕ that minimizes $preserved(\phi)$ that is, the one that minimizes our estimation of the size of the explored search space. Next section provides more details about the MCS selection algorithm.

3.3 MCS selection algorithm

At each search node, the MCS selection procedure develops a tree of partial MCSs where the current partial MCS ϕ is extended adding one resource requirement in each child node. By definition of $preserved(\phi)$, it is clear that $\phi' \subset \phi \Rightarrow preserved(\phi') \leq preserved(\phi)$. Thus, if ϕ^* is the best MCS found so far, once a partial MCS ϕ has been reached such that $preserved(\phi^*) \leq preserved(\phi)$, the sub-tree of the MCS selection tree rooted at ϕ can be abandoned.

The algorithm for selecting and branching on a MCS is more precisely described in Algorithm 2 using the following notations: $id(u)$ is a unique index associated with resource requirement u used to break ties. $Unranked(u)$ represents all the resource requirements v that can possibly overlap u given the current temporal constraints, and that are such that $q(v) < q(u)$ or $q(v) = q(u)$ and $id(v) < id(u)$. ϕ is the (partial) MCS that is currently being extended. A (partial) MCS is represented by a list of resource requirements: $\phi = (u_1, \dots, u_k)$. We denote $u_k = Last(\phi)$ and define the operator $\oplus$ as follows: $\phi \oplus u = (u_1, \dots, u_k, u)$; $q = q(\phi)$ is the consumption of the current (partial) MCS; $p = preserved(\phi)$

¹Note that this is of course only a rough estimate and in particular, the estimated fraction $preserved(\phi)$ can be greater than 1.

is the preserved search space so far in the current (partial) MCS; ϕ^* is the best MCS so far and $p^* = preserved(\phi^*)$ is the preserved search space of the best MCS so far. The procedure at line 1 calls the MCS rating and selection process on each resource. At line 6, to rate and select MCSs for a given resource, the procedure first sorts the relevant sets of requirements v at lines 7 and 9 by decreasing order of $q(v)$, using $id(v)$ to break ties in order to ensure that each MCS is scanned only once, starting with the smallest MCSs, that is, the ones containing the most greedy requirements. The procedure at line 23 returns TRUE if and only if a given requirement u can possibly overlap all the requirements of a partial MCS given the current temporal network. The procedure at line 28 computes the incremental increase of preserved space due to the insertion of a new requirement u in the current MCS ϕ . The value of $preserved$ has been described in section 3.2. The main recursive function for selecting MCSs is described at line 12.

Algorithm 2 MCS selection algorithm

```

1: procedure SELECT_MCS
2:    $p^* \leftarrow +\infty$ 
3:   for  $R$  in  $\mathcal{R}$  do
4:     SELECT_MCS( $R$ )
5:   return  $\phi^*$ 

6: procedure SELECT_MCS( $R$ )
7:   Sort  $U(R)$  by decreasing  $q$ 
8:   for all  $u$  in  $U(R)$  do
9:     Sort  $Unranked(u)$  by decreasing  $q$ 
10:    for  $u$  in  $U(R)$  do
11:      RSELECT_MCS( $R, (u), q(u), 0$ )

12: procedure RSELECT_MCS( $R, \phi, q, p$ )
13:   if  $q > Q(R)$  then  $\triangleright \phi$  is a MCS
14:     if  $p < p^*$  then  $\triangleright \phi$  is the best MCS so far
15:        $p^* \leftarrow p$ 
16:        $\phi^* \leftarrow \phi$ 
17:   else  $\triangleright \phi$  needs to be extended
18:      $u \leftarrow Last(\phi)$ 
19:     for all  $v$  in  $Unranked(u)$  do
20:       if IS_UNRANKED( $v, \phi$ ) then
21:          $dp \leftarrow DELTA\_PRESERVED(v, \phi)$ 
22:         RSELECT_MCS( $R, \phi \oplus v, q + q(v), p + dp$ )

23: procedure IS_UNRANKED( $u, \phi$ )
24:   for all  $v$  in  $\phi$  do
25:     if  $u \preceq v$  or  $v \preceq u$  then
26:       return FALSE
27:   return TRUE

28: procedure DELTA_PRESERVED( $u, \phi$ )
29:    $dp \leftarrow 0$ 
30:   for all  $v$  in  $\phi$  do
31:      $dp \leftarrow dp + preserved(u, v) + preserved(v, u)$ 
32:   return  $dp$ 

```

The best MCS ϕ^* that has been scanned by the above pro-

cedure is selected as the one to be solved at the current search node. This MCS is simplified using Algorithm 1 and the search explores all of its resolvers $\rho \in \text{Res}(\phi^*)$ in the child nodes by decreasing order of $\text{preserved}(\rho)$. This order has no effect when the schedule is not feasible as in this case the complete search tree needs to be explored but it helps finding a solution quicker when a solution exists.

4 Self-adapting shaving

Shaving techniques [Torres and Lopez, 2000] provide a good framework for strengthening constraint propagation and avoiding late failures to be discovered in the search tree. They are all based on the following principle: if adding a constraint C in the current node of the search leads to a failure of the propagation, then, constraint $\neg C$ can be inferred. Due to the cost of propagating a constraint C and the potential number of constraints C to try to shave on, shaving techniques are in general computationally expensive.

To improve the pruning of the search tree, we implemented the following shaving technique based on MCSs. If a MCS ϕ with resolvers $\text{Res}(\phi) = \{\rho_1, \dots, \rho_k, \rho_{k+1}\}$ is such that $\forall i \in [1..k]$, adding ρ_i in the current schedule leads to a failure of the propagation, then ρ_{k+1} can be inferred. The complexity for shaving a given MCS ϕ of size n is thus in $O(n^2P)$ where P is the cost of full constraint propagation at the current node. Potentially, there is of course an exponential number of MCSs to shave on at each search node and we can expect that many of those MCS do not allow inferring any precedence constraint. An idea to speed-up the shaving process is thus to only try shaving on a subset of MCSs for which the probability to infer a precedence constraint is greater than a given threshold α . Parameter α is an input of the shaving algorithm. We can roughly estimate that the probability that adding a precedence constraint $x \preceq y$ in the current schedule will lead to a failure of the propagation is proportional² to $1 - \text{preserved}(x \preceq y)$. If $\rho_m = \arg\max_{\rho \in \text{Res}(\phi)} \text{preserved}(\rho)$ is the resolver of the MCS ϕ with maximal preserved search space, we are interested in the MCSs that get a high probability that all their resolvers but ρ_m will fail, that is, if we assume all the probabilities are independent, the ones such that $\prod_{\rho \in \text{Res}(\phi) \setminus \{\rho_m\}} (1 - \text{preserved}(\rho))$ is greater than a given threshold. For those MCSs, if the threshold is close enough to 1, we can assume that $\text{preserved}(\rho)$ is small enough so that the first order approximation $\prod_{\rho \in \text{Res}(\phi) \setminus \{\rho_m\}} (1 - \text{preserved}(\rho)) \approx 1 - \sum_{\rho \in \text{Res}(\phi) \setminus \{\rho_m\}} \text{preserved}(\rho)$ is reasonable.

To summarize, we thus only consider for shaving those MCS scanned by the procedure described in Algorithm 2 that are such that $\text{preserved}(\phi) - \text{preserved}(\rho_m) \leq \beta$, β being a threshold. The computation of this criterion only adds a negligible overhead related with the maintenance of ρ_m for each MCS in the MCS selection procedure. Due to the numerous approximations, β is not taken to be constant (theoretically equal to $1 - \alpha$). The threshold β is computed by

²Note that this estimation is exact at the extreme points when $\text{preserved}(x \preceq y) = 0$ (propagation will fail for sure) and when $\text{preserved}(x \preceq y) = 1$ (propagation cannot fail because $x \preceq y$ has already been discovered given the current domains of x and y).

self-adaptation in such a way that on average, among the last h shaving attempts, αh lead to the inference of a new precedence. Whenever the number of successful shaving s among the last h ones deviates from αh , the parameter β is adapted accordingly: if $s < \alpha h$, β is decreased by $\epsilon\beta$ and if $s > \alpha h$, β is increased by $\epsilon\beta$. For all our experiments with shaving, we took $h = 20$, $\alpha = 0.75$, $\epsilon = 0.01$ and start with $\beta = 1$.

5 Experimental evaluation

The approach has been implemented on top of ILOG SCHEDULER 6.1 using the *timetable*, *disjunctive*, *edge-finder*, *precedence energy* and *balance* constraints [ILOG, 2005]. All the experiments described in this section were run on a Dell Latitude D600 laptop, 1.4 GHz. Detailed results such as individual lower bounds for each problem instance and new optimal solutions are available at <http://scheduler.ilog.fr/>.

5.1 Results on general RCPSP

We evaluated our approach on the instances of the PSPLIB [Kolisch and Sprecher, 1996] with 60, 90 and 120 activities (resp. sets J60, J90, J120). For each instance, we solve the feasibility problem of finding a schedule with a makespan lower than T , starting with a legal lower bound for T^3 and incrementing T until either the problem is shown to be feasible (in this case, T is the optimal makespan) or a given time limit for solving the problem with makespan T is exceeded (in this case, T is a legal lower-bound but the search stops without providing any legal upper-bound).

In a first series of experiments, we use the basic search described in section 3 without shaving with a time limit of 300s. The previous best lower and upper bounds we compare with are the ones reported in the PSPLIB together with the recent improvements on the J60 instances reported in [Baptiste and Demassey, 2004]. The results are summarized on Table 1 with the following columns:

- #O** : number of instances previously open
- #I** : number of improved lower bounds (% of #O)
- AGR**: average gap (distance from the lower to the upper bound) reduction when a bound is improved
- #C** : number of closed instances (% of #O)

Inst.	#O	#I	(%I)	AGR	#C	(%C)
J60	98	39	(39.8%)	62.9%	21	(21.4%)
J90	129	51	(39.5%)	58.4%	26	(20.2%)
J120	390	88	(22.6%)	47.0%	38	(9.7%)
ALL	617	178	(28.8%)	53.0%	85	(13.8%)

Table 1: Results on RCPSP without self-adapting shaving with a time-limit of 300s

Out of the 617 previously open instances, we are able to improve 178 lower-bounds with an average gap reduction of 53% and to close 85 instances.

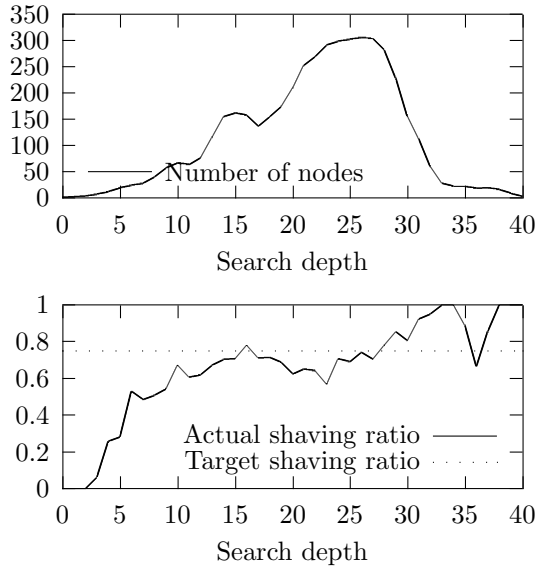
To show the effect of self-adapting shaving, we run a version of our approach using self-adapting shaving with the

³For instance the lower-bound of the PERT of temporal constraints

Inst.	#O	#I	(%I)	AGR	#C	(%C)
J60	98	40	(40.8%)	62.6%	22	(22.4%)
J90	129	52	(40.3%)	58.3%	26	(20.2%)
J120	390	90	(23.1%)	47.3%	38	(9.7%)
ALL	617	182	(29.5%)	53.1%	86	(13.9%)

Table 2: Results on RCPSP with a self-adapting shaving and a time-limit of 300s

same time-limit of 300s. The results are summarized on Table 2. Out of the 617 previously open instances, we are able to improve 182 lower-bounds with an average gap reduction of 53.1% and to close 86 instances. The main conclusion is that, within the same time-limit, the addition of self-adapting shaving slightly increases the performances. The two curves below respectively show, on a particular instance (J60_5.2), the number of search nodes with a given depth in the search tree and, for each node depth, the ratio between the number of selected MCSs that effectively lead to the inference of a new precedence and the total number of MCSs selected for shaving. One clearly see that in the region of the search space where most of the nodes are concentrated (between depths 10 and 35), this ratio is effectively close to the target ratio of 0.75. In this instance, 4667 nodes were explored, 4843 MCSs were selected for shaving and among them, 3291 effectively lead to the inference of a new precedence.



Given that self-adapting shaving slightly improves the performances within the same time limit of 300s, we used this configuration with an extended time-limit of 1800s. The results are summarized on Table 3. Out of the 617 previously open instances, we improve 193 lower-bounds (that is more than 31% of the previously open instances) with an average gap reduction of 61.1% and close 96 instances (that is more than 15% of the previously open instances).

5.2 Results on open-shop problems

Open-shop problems can be represented as a special cases of RCPSP where all resources have a unit capacity and additional unary resources are used to model the fact that activities of the same job cannot not overlap. We tested our

Inst.	#O	#I	(%I)	AGR	#C	(%C)
J60	98	44	(44.9%)	72.4%	26	(26.5%)
J90	129	54	(41.9%)	66.0%	30	(23.3%)
J120	390	95	(24.4%)	48.1%	40	(10.3%)
ALL	617	193	(31.3%)	61.1%	96	(15.6%)

Table 3: Results on RCPSP with a self-adapting shaving and a time-limit of 1800s

Instance	UB	Optim.	Instance	UB	Optim.
gp06-03	1255	1255*	gp09-02	1112	1110*
gp06-07	1290	1290*	gp09-03	1117	1115*
gp06-09	1243	1243*	gp09-06	1093	1093*
gp06-10	1254	1254*	gp09-07	1097	1090*
gp07-01	1159	1159*	gp09-08	1106	1105*
gp07-02	1185	1185*	gp09-09	1126	1123*
gp07-04	1167	1167*	gp09-10	1120	1110*
gp07-06	1193	1193*	gp10-01	1099	1093*
gp07-08	1180	1180*	gp10-02	1099	1097*
gp07-09	1220	1220*	gp10-03	1081	1081*
gp08-02	1135	1135*	gp10-04	1089	1077*
gp08-04	1154	1153*	gp10-05	1080	1071*
gp08-06	1116	1115*	gp10-06	1072	1071*
gp08-07	1126	1126*	gp10-07	1081	1079*
gp08-08	1148	1148*	gp10-08	1098	1093*
gp08-10	1161	1161*	gp10-09	1120	1112*
gp09-01	1135	1129*	gp10-10	1092	1092*

Table 4: Results on open-shop with a self-adapting shaving and a time-limit of 5s

approach, with the same settings as in previous section on the open-shop problems instances proposed in [Gu  ret and Prins, 1999]. Those instances are considered to be very hard instances of open-shop problems and serve as classical benchmark in open-shop scheduling (see for instance recent work in [Blum, 2005]). The benchmark consists of 80 instances ranging from 3 jobs $\times$ 3 machines problems until 10 jobs $\times$ 10 machines problems. Out of these 80 problems, 34 instances are still open. Using our approach, we were able to close all those instances in less than 5s CPU time. The optimal makespan for the 34 previously open instances is summarized on Table 4 where the column *UB* corresponds to the currently best known upper-bound for which no optimality proof did exist.

We also experimented with the open instances of the benchmark of [Brucker *et al.*, 1997]. We closed 3 of these 6 open instances: namely *j8-per0-2* (optimal makespan: 1052), *j8-per10-0* (optimal makespan 1017) and *j8-per10-1* (optimal makespan 1000).

5.3 Results on cumulative jobshop problems

We tested our approach, with the same settings, on the cumulative job-shop problem benchmark described in [Nuijten, 1996]. These instances are derived from classical jobshop scheduling problems by multiplying the number of jobs (and thus the number of activities) and the capacity of the resources by a given factor ($\times 2$ or $\times 3$). Our results are summarized on Table 5 where *LB* is the lower bound using the consistency checking described in [Nuijten, 1996] and *NewLB* the new lower bound of our approach. We were able to close the *ft06* $\times$ 2 and *ft06* $\times$ 3 instances as well as to improve 12 lower bounds out of these 38 open instances.

Instance	LB	New LB	Instance	LB	New LB
ft06×2	53	55*	ft06×3	53	55*
ft10×2	835	837	ft10×3	828	828
la03×2	593	593	la03×3	590	590
la04×2	572	572	la04×3	570	570
la16×2	888	892	la16×3	884	887
la17×2	754	754	la17×3	753	753
la18×2	783	803	la18×3	776	783
la19×2	731	756	la19×3	724	740
la20×2	830	849	la20×3	829	842
la21×2	1017	1017	la21×3	1010	1012
la22×2	913	913	la22×3	913	913
la24×2	885	885	la24×3	884	884
la25×2	907	907	la25×3	903	903
la29×2	1117	1117	la29×3	1116	1116
la36×2	1229	1229	la36×3	1227	1227
la37×2	1378	1378	la37×3	1370	1370
la38×2	1092	1092	la38×3	1087	1087
la39×2	1221	1221	la39×3	1221	1221
la40×2	1180	1180	la40×3	1176	1176

Table 5: Results on cumulative job-shop with a self-adapting shaving and a time-limit of 1800s

6 Conclusions

We presented a simple complete search procedure implemented on top of classical constraint propagation algorithms and applied it to resource constrained project scheduling problems. In average, this approach outperforms the best algorithms for finding lower bounds on those scheduling problems, even with a time limit of 300s per optimization step⁴. Using this approach in conjunction with a self-adapting shaving procedure, we were able to close more than 15% of the previously open problems of the PSPLIB and improve more than 31% of the best known lower bounds. What is even more remarkable is that this very same approach allows closing all the hard open-shop instances of [Guéret and Prins, 1999] in less than 5s CPU time although the approach was not particularly designed to tackle disjunctive scheduling and does not exploit the open-shop nature of the problems.

The understanding of why our method works so well on the instances of the PSPLIB and on many open-shop problems would deserve a deeper study. From one hand, if the problems are highly cumulative, our approach is clearly limited by the explosion of the number of MCSs to consider. From the other hand, when problems are highly disjunctive, we could expect other approaches dedicated to disjunctive scheduling to work better. A first possible explanation could be a good fit between our approach and the “disjunctivity” degree of the hard instances of the PSPLIB as suggested by some recent work [Garaix *et al.*, 2005]. A result of this study could be some kind of hybridizing of MCS-based search with techniques more adapted to highly cumulative problems, MCS-based search being restricted to the resolution of MCSs with small preserved search space (thus small MCSs) at the top of the search tree. A second direction for future work is the generalization of the notion of *self-adapting* shaving introduced in this paper to other shaving techniques in scheduling.

⁴When this time limit is exceeded at an optimization step, usually, the previous steps were fairly quick so that the overall time for computing the lower bound is close to 300s.

References

- [Baptiste and Demassey, 2004] P. Baptiste and S. Demassey. Tight LP bounds for resource constrained project scheduling. *OR Spectrum*, 26:251–262, 2004.
- [Blum, 2005] C. Blum. Beam-ACO - hybridizing ant colony optimization with beam search: an application to open-shop scheduling. *Computers & Operations Research*, 32(6):1565–1591, 2005.
- [Brucker and Knust, 2000] P. Brucker and S. Knust. A linear programming and constraint propagation-based lower bound for the RCPSP. *European Journal of Operational Research*, 127:355–362, 2000.
- [Brucker *et al.*, 1997] P. Brucker, J. Hurink, B. Jurisch, and B. Wöstmann. A branch & bound algorithm for the open-shop problem. *Discrete Applied Mathematics*, 76:43–59, 1997.
- [Demeulemeester and Herroelen, 2002] E. Demeulemeester and W. Herroelen. *Project scheduling - A research handbook*. Kluwer Academic Publishers, 2002.
- [Garaix *et al.*, 2005] T. Garaix, C. Artigues, and S. Demassey. Bornes basées sur les ensembles interdits pour le problème d’ordonnancement de projet à moyens limités. In *ROADEF’2005*, 2005.
- [Guéret and Prins, 1999] C. Guéret and C. Prins. A new lower bound for the open-shop problem. *Annals of Operations Research*, 92:165–183, 1999.
- [Hartmann and Kolisch, 2005] S. Hartmann and R. Kolisch. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem: An update. *European Journal of Operational Research*, 2005.
- [ILOG, 2005] ILOG. ILOG SCHEDULER 6.1 Reference Manual, 2005. <http://www.ilog.com/>.
- [Kolisch and Sprecher, 1996] R. Kolisch and A. Sprecher. PSPLIB - A project scheduling problem library. *European Journal of Operational Research*, 96:205–216, 1996.
- [Laborie and Ghallab, 1995] P. Laborie and M. Ghallab. Planning with Sharable Resource Constraints. In *IJCAI-95*, pages 1643–1649, 1995.
- [Laborie, 2003] P. Laborie. Algorithms for propagation resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143:151–188, 2003.
- [Nuijten, 1996] W. Nuijten. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operational Research*, 90(2):269–284, 1996.
- [Torres and Lopez, 2000] P. Torres and P. Lopez. Overview and possible extensions of shaving techniques for Job-Shop problems. In *CP-AI-OR’2000*, pages 181–186, 2000.
- [Vilain and Kautz, 1986] M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *Fifth National Conference on Artificial Intelligence*, pages 377–382, 1986.

Three Truth Values for the SAT and MAX-SAT Problems

Frédéric Lardeux and Frédéric Saubion and Jin-Kao Hao

LERIA, University of Angers,

2 Bd Lavoisier,

F-49045 Angers Cedex 01

{lardeux,saubion,hao}@info.univ-angers.fr

Abstract

The aim of this paper is to propose a new resolution framework for the SAT and MAX-SAT problems which introduces a third truth value *undefined* in order to improve the resolution efficiency. Using this framework, we have adapted the classic algorithms Tabu Search and Walksat. Promising results are obtained and show the interest of our approach.

1 Introduction

The satisfiability problem (SAT) [Garey and Johnson, 1979] consists in finding a truth assignment that satisfies a well-formed Boolean expression. An instance of the SAT problem is then defined by a set of Boolean variables (also called atoms) $\mathcal{X} = \{x_1, \dots, x_n\}$ and a Boolean formula $\phi: \mathcal{B}^n \rightarrow \mathcal{B}$. As usual, the formula ϕ is supposed to be in conjunctive normal form (CNF) (i.e., it is a conjunction of clauses where a clause is a disjunction of literals¹). The formula is said to be satisfiable if there exists an assignment (i.e., a function $A: \mathcal{X} \rightarrow \mathcal{B}^n$) satisfying ϕ and unsatisfiable otherwise. The search space $\mathcal{S}$ corresponds to the set of all possible assignments. ϕ is obviously satisfied if all of its clauses are satisfied. In this context, the maximum satisfiability problem (MAX-SAT) corresponds to the minimization of the number of false clauses.

Two classes of methods are used to solve SAT and MAX-SAT problems, namely exact and approximate methods.

Exact methods are able to find all the solutions of an instance or, if there is no solution, to prove its unsatisfiability. These methods are generally based on the Davis-Putnam-Logemann-Loveland procedure [Davis *et al.*, 1962] which explores a binary search tree, building incrementally truth assignments. They provide very good results but are not suitable for the MAX-SAT problem. Other exact methods, generally based on Branch and Bound (B&B) algorithms [Borchers and Furman, 1999] have been designed to handle MAX-SAT, but their performances are often limited for large instances.

Approximate methods are mainly based on local search and evolutionary algorithms. We focus here on local search algorithms which have been widely studied in the SAT community [Selman *et al.*, 1992; 1994; Spears, 1996; Mazure *et*

al., 1997; Hirsch and Kojevnikov, 2001]. As usual, these algorithms rely on the fundamental characteristics of local search: the exploration of the search space is achieved by moving from an element to one of its neighbors, these moves being performed according to specific heuristics. Therefore, they expect to minimize, on the assignments space, a function corresponding to the number of false clauses. They are thus naturally designed for the MAX-SAT problem and allow one to handle large instances.

The efficiency of such algorithms is strongly related to their capacities to finely exploit specific areas of the search space when needed (i.e., intensify the search) and to widely explore the space by moving to different promising areas (i.e., diversify the search). These aspects are controlled through parameters to ensure a good tradeoff between diversification and intensification. But, actually these two notions are not really well defined and therefore, parameters tuning remains a key point when using such algorithms.

We should remark that exact and approximate methods do not use the same representations for their search space. Their hybridizations [Mazure *et al.*, 1998; Prestwich, 2000; Habet *et al.*, 2002; Prestwich, 2004] are not trivial since exact methods work on partial assignments incrementally completed by the resolution process while approximate algorithms explore the set of all possible complete assignments. A partial assignment is an assignment where some variables are not yet valued and can be thus considered as the representation of a set of complete assignments. From a local search point of view, the use of such partial assignments could lead to new diversification and intensification processes. Indeed, the diversification of the search could be achieved by moving over large sets of complete assignments induced by given partial assignments, while the valuation of some variables of such partial assignments could be considered as a way to focus on a more restricted part of the search space.

The purpose of this paper is to define a uniform model for these two types of assignments in order to precisely define and study the fundamental mechanisms of local search for SAT and MAX-SAT resolutions. This framework allows us to propose a new local search scheme which provides a uniform control of the search and which can be introduced in well known local search procedures: Tabu Search and Walksat.

¹A literal is a variable or its negation.

2 A 3-Valued Framework

We propose a new resolution framework for the SAT and the MAX-SAT problems which introduces a third truth value *undefined* in order to improve the resolution efficiency. Within this framework, local search methods will be extended in order to take into account partial assignments and, therefore, several notions and logical rules have to be redefined.

2.1 Standard Local Searches

In the classic SAT and MAX-SAT context, the search space S is the set of all possible complete truth assignments. The objective function to maximize corresponds to the number of satisfied clauses given by the *standard_eval* function:

$$\begin{aligned} \text{standard_eval: } S &\rightarrow \mathbb{N} \\ A &\mapsto |\{c | \text{sat}(A, c) \wedge c \in \phi\}| \end{aligned}$$

where $\text{sat}(A, c)$ means that the clause c is satisfied by the assignment $A \in S$ and $|E|$ is the cardinality of the set E .

The moves are clearly possible flips of the values of a given assignment. The flip of a variable i in an assignment A is the swap of its truth value (T (*true*) to F (*false*) or F to T). The best flip is selected thanks to a *standard_choose* function which returns the variable whose flip provides the best improvement (i.e., maximizes the number of false clauses which become true after the flip minus the number of satisfied clauses which become false).

A naive local search algorithm for SAT and MAX-SAT consists in selecting the best move at each step (using *standard_choose*). It stops either when a solution is found or when a maximum number of steps has been reached. Then, it returns the best assignment found (w.r.t. *standard_eval*). Specific control heuristics can be added to improve the efficiency of the search, such as random walks and other noise strategies [Selman *et al.*, 1994].

2.2 Introducing an undefined Value

Our purpose is to provide a uniform framework to represent complete and partial assignments. Concerning complete assignments, with classical *true* and *false* values, the logical interpretation rules are well-known but, with partial assignments, non valued variables appear and we decide to introduce a third truth value to represent them.

We add a truth value U (*undefined*) and consider the set of truth values $\mathcal{T} = \{U, T, F\}$ in our *3-valued framework*. This new value induces some changes in the standard logical interpretation rules. Two approaches can be used to define these rules: an optimistic approach and a pessimistic approach.

- The *optimistic approach* is the most commonly used. Indeed, all the exact methods exploring a search tree are based on this approach. A clause is considered undefined if none of its literals is true and at least one of its literals is undefined. This approach corresponds to the following simplification rules:

$$\begin{aligned} T \vee U &\rightarrow T \\ F \vee U &\rightarrow U \\ U \vee U &\rightarrow U \end{aligned}$$

- We propose here an alternative *pessimistic approach* which relies on the fact that as soon as a clause has a false literal and no true literal, it is considered as false. An undefined clause must have all its literals set to undefined. This approach corresponds to the following rules:

$$\begin{aligned} T \vee U &\rightarrow T \\ F \vee U &\rightarrow F \\ U \vee U &\rightarrow U \end{aligned}$$

Our search structure is now defined and we may study the precise behavior of local search algorithms on this structure with three truth values.

2.3 Local Search Transitions

We define a partial order $\sqsupseteq$ on the set $\mathcal{T}$ such as $U \sqsupseteq T$ and $U \sqsupseteq F$. In this context, given a SAT problem with n variables, our search space will be the set $\mathcal{T}^n$. The ordering relation $\sqsupseteq$ can be naturally extended to $\mathcal{T}^n$: $(x_1, \dots, x_n) \sqsupseteq (y_1, \dots, y_n)$ if and only if $\exists i, x_i \sqsupseteq y_i$ and $\nexists j, y_j \sqsupseteq x_j$. We consider the partial ordering $(\mathcal{T}^n, \sqsupseteq)$ with the greatest element $\top = (U, \dots, U)$. This structure will now be used to precisely describe the behavior of local search algorithms. We consider a basic local search move² as a transition $A \rightarrow A'$ where $A, A' \in \mathcal{T}^n$. Since we only consider here a classical neighborhood we require that transitions $(x_1, \dots, x_n) \rightarrow (y_1, \dots, y_n)$ satisfy $\exists i, x_i \neq y_i$ and $\forall j \neq i, x_j = y_j$. This neighborhood corresponds to a hamming distance equal to 1 between two neighbors, which is related to the *flip* function.

On the one hand, we first consider the assignment landscape point of view. The 3-valued framework allows us to handle both partial and complete assignments and, as mentioned above, a partial assignments is seen as a representation of a set of smaller assignments (w.r.t. $\sqsupseteq$). Therefore, diversification consists in considering larger sets of elements of the search space while intensification focus on fewer elements. The transition corresponds to a layer exploration when the number of undefined variables remains the same.

- **Diversification:** $A \rightarrow A'$ with $A' \sqsupseteq A$
- **Intensification:** $A \rightarrow A'$ with $A \sqsupseteq A'$
- **Layer Exploration:** $A \rightarrow A'$ with $A \not\sqsupseteq A'$ and $A' \not\sqsupseteq A$

On the other hand, we have to take into account the evaluation of the assignments. The introduction of the third truth value requires us to redefine the *standard_eval* function and the *standard_choose* function of section 2.1, since the number of undefined clauses must be taken into account. The new evaluation function *eval* returns the number of true clauses as well as the number of undefined clauses. In this context, an assignment is better than another if it satisfies more clauses. If the number of true clauses is equal for two assignments, the one generating the greatest number of undefined clauses will be considered as the best assignment.

$$\begin{aligned} \text{eval: } S &\rightarrow (\mathbb{N}, \mathbb{N}) \\ A &\mapsto (|\{c | \text{sat}(A, c) \wedge c \in \phi\}|, \\ &\quad |\{c | \text{undefined}(A, c) \wedge c \in \phi\}|) \end{aligned}$$

²With three truth values, the concepts of flip is not valid anymore. We prefer to use the concept of move, step or transition.

where $undefined(A, c)$ means that the clause c is undefined by the assignment $A \in \mathcal{S}$.

Using this function, we may define Sol_ϕ , the set of all the solutions for a formula ϕ : $Sol_\phi = \{A | eval(A) = (k, 0), A \in \mathcal{T}^n\}$ with $k = |\{c | c \in \phi\}|$.

We define the order $>_{eval}$ as the lexicographic extension $(>, >)$ of the order $>$ on the pair constructed by the $eval$ function. We may now consider the transitions from this evaluation point of view.

- **Improve:** $A \rightarrow A'$ with $A' >_{eval} A$
- **Deteriorate:** $A \rightarrow A'$ with $A >_{eval} A'$
- **Preserve:** $A \rightarrow A'$ with $eval(A') = eval(A)$

Using these concepts, a move is now characterized from each point of view: for instance, a given move may correspond to a diversification and improve transition.

These notions are illustrated in example 1. Indeed, in our 3-valued framework, both exact and approximate methods can be modeled. The search trees used by Davis-Putnam-Logemann-Loveland like procedures are sets of intensification and diversification transitions whereas classic local search algorithms perform sets of improving, deteriorating and preserving transitions at the lowest layer (with no undefined value).

Example 1: $(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b)$

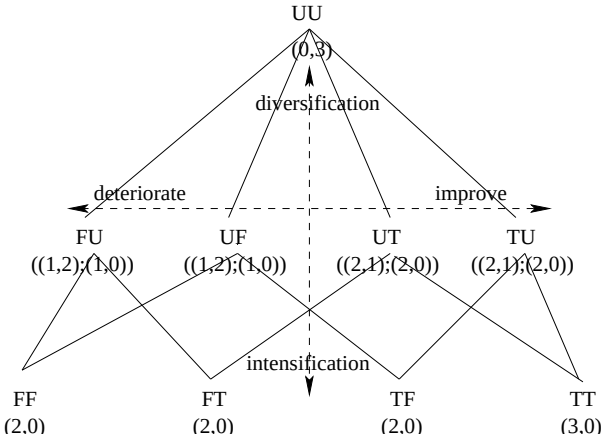


Figure 1: The 3-valued framework. All the elements of $\mathcal{T}^n$ are represented w.r.t. the ordering relation $\sqsubset$. A diversification transition increases the number of undefined variables whereas an intensification transition decreases this number. In this scheme, complete assignments are at the bottom and $\top = (UU)$ is on top. When the number of undefined variables remains stable, the elements are sorted by the order $>_{eval}$. The valuations of the $eval$ function are given as a couple $(v_1; v_2)$ where v_1 (resp. v_2) corresponds to the evaluation of the current assignment under the optimistic (resp. pessimistic) approach. This couple is reduced to a single evaluation if $v_1 = v_2$. The best element is on the right side and the worst on the left side. Therefore, in this figure, best solutions are on rightmost bottom positions.

At this stage, one has to consider possible combinations of these transitions, issued from these two points of view, to build a general local search process.

The purpose of the search process is to increase the number of true clauses. When this is not possible, it is then preferable to increase the number of undefined clauses and thus to reduce the number of false clauses. We propose a new *choose* function which selects a variable and a change of its value which maximizes the number of true clauses and minimizes the number of false clauses (increasing the number of undefined clauses if necessary).

At this step, the main components of a local search algorithm have been described. We will now study their behaviour and combination.

2.4 Evaluation of the Logical Rules

We want to measure the effect of the logical interpretation rules, presented in section 2.2, on the search process. As described before, a 3-valued local search (3LS) algorithm may return now partial assignments which represent indeed sets of assignments. To evaluate the quality of these partial assignments, it is necessary to explore exhaustively the corresponding subset in order to find the best complete assignment that it contains. This can be achieved thanks to a Branch and Bound procedure (B&B) which performs an exhaustive search by exploring a restricted search tree whose root is the partial assignment provided by the 3LS algorithm.

Table 1 compares the effects of the different logical rules. Tests are realized on standard random 3-sat instances with 500, 1000 and 2000 variables (F500.cnf, F1000.cnf and F2000.cnf). The 3LS algorithm used for these tests is a 3-valued Tabu Search (3TS-BB, see next section). The algorithm runs until 5000 iterations are performed. The number of undefined variables (*nb. var. U*), the number of clauses with one, two and three undefined variables ((U, UU, UUU)) and the quality $((T, U, F)$ after 3TS-BB) are memorized for the last partial assignment found. To estimate the potential quality of the partial assignment found by 3TS-BB, a B&B algorithm is executed and provides the best complete assignment $((T, F)$ after B&B) by assigning the undefined variables.

Inst.	Observations	Optimistic	Pessimistic
500 var.	(T,U,F) after 3TS-BB	(2139,1,10)	(2145,0,5)
	nb. var. U	20	18
	(U,UU,UUU)	(188,2,0)	(177,4,0)
	(T,F) after B&B	(2140,10)	(2145,5)
1000 var.	(T,U,F) after 3TS-BB	(4227,5,18)	(4233,0,17)
	nb. var. U	33	40
	(U,UU,UUU)	(356,5,0)	(394,16,0)
	(T,F) after B&B	(4232,18)	(4234,16)
2000 var.	(T,U,F) after 3TS-BB	(8455,1,44)	(8465,0,35)
	nb. var. U	69	78
	(U,UU,UUU)	(662,16,0)	(774,20,0)
	(T,F) after B&B	(8456,44)	(8465,35)

Table 1: Comparisons between the two approaches.

Table 1 shows a clear dominance of the pessimistic interpretation. The quality of the partial assignment found during

the 3TS-BB is better since the number of true clauses found after B&B is greater than the corresponding number obtained with the optimistic interpretation.

2.5 Impact of Transitions in the Search Process

The optimistic and the pessimistic interpretations do not involve the same behaviour w.r.t. the different combinations of transitions characterizing a move, as described in section 2.3. Table 2 presents the evolution of the number of true, undefined and false clauses (T,U,F) for the optimistic (opt.) and the pessimistic (pes.) approaches and all the combinations of transitions excepted for the “preserve” transition since no change is performed in this case.

	Diversification			Intensification			Layer exploration		
	T	U	F	T	U	F	T	U	F
Imp. (opt.)	=	>	<	>	<	>	>	~	~
pes.)	=	>	<	>	<	>	>	=	<
Det. (opt.)	<	>	<	=	<	>	<	~	~
pes.)	<	>	~	=	<	>	<	=	>

Table 2: Possible evolutions of the number of true, undefined and false clauses (T,U,F) due to the different combinations of transitions. (>: increase, ≥: increase or remain stable, =: remain stable, ≤: decrease or remain stable, <: decrease and ~: increase, decrease or remain stable)

We may assess common properties for the optimistic and pessimistic approaches. When applying an intensification transition, it is neither possible to create undefined clauses nor to delete true clauses. Concerning the diversification transition, true clauses cannot be created and undefined clauses cannot be deleted. Specific properties related to the optimistic and pessimistic interpretations can also be depicted: using the optimistic approach, false clauses cannot be generated during a diversification transition while in the pessimistic case, undefined clauses cannot be created nor deleted during the “Layer exploration”.

3 3-valued Framework used in Well-known Algorithms

Tabu Search (TS) [Glover and Laguna, 1997; Mazure *et al.*, 1997] and Walksat [Selman *et al.*, 1994] are two well-known local search algorithms for SAT and MAX-SAT. To show the interest of the 3-valued framework, we have adapted these two algorithms in order to obtain 3TS-BB and 3Walksat-BB. The *choose* and the *eval* functions (2.3) were used to transform the two algorithms in their 3-valued versions.

3.1 3TS-BB

The 3-valued Tabu Search (3TS-BB) is a local search meta-heuristics based on standard Tabu Search. Its aim is to maximize the number of true clauses and minimize the number of false clauses. The exploration of the search space is achieved by moving from an assignment to an other selected by the *choose* function. These moves are guided by the *eval* function which evaluates their benefit. In order to avoid problems induced by local optima, 3TS-BB uses a tabu list containing informations on previously visited configurations to forbid some possible loops in the search process. The length of the

tabu list is set to 10% of the number of variables (empirical result inspired by [Mazure *et al.*, 1997]). One expects 3TS-BB to return a complete assignment to answer to the MAX-SAT problem. Then, we decide to regularly complete the partial assignments with a B&B procedure in order to generate a complete assignment. This completion is realized every s moves (empirically fixed to 5000). To reduce the execution time, B&B must not be used when there are too many undefined variables in the partial assignments. We impose that the maximum number of unassigned variables should not to be greater than a bound b (empirically fixed to 100). The search stops when a solution is found or when 10^6 moves are executed.

3.2 3Walksat-BB

The general process of the 3-valued Walksat (3Walksat-BB) uses similar principles. Like the standard Walksat, 3Walksat-BB is a randomized local search algorithm, which tries to determine the best move by randomly choosing an unsatisfied clause and selecting one of its variable thanks to the *choose* function. We use as basis the version v41 of Walksat. The search is composed of 10 tries. Each try starts with a random assignment and after 10^5 moves, the best assignment found is completed with a classic B&B, if it is not a complete assignment. When a solution is found, the search stops. Like Walksat, 3Walksat-BB uses the “novelty” heuristic [Selman *et al.*, 1994] with a noise set to 0.5 (its default value).

Remark: It is important to note that, contrary to the standard local search algorithms, the 3-valued algorithms may start their searches with the $\top$ assignment ($U, \dots, U$). The influence of the initial assignment in the results is then avoided.

4 Experimental Results

Due to the approximate and non-deterministic nature of TS, 3TS-BB, Walksat and 3Walksat-BB, each algorithm runs 20 times on each benchmark. Tests are realized on a cluster with Linux and Alinka (5 nodes each of them with 2 CPU Pentium IV 2.2 GHz and 1 Gb of RAM) used sequentially. The maximum number of allowed local search steps is 10^6 . For our algorithms, we consider that a backtrack performed during the B&B procedure has the same cost than a local search step.

All studied instances (Table 3) have been used for SAT competition 2003 [Simon *et al.*, 2003] and contain satisfiable and unsatisfiable instances. They come from different families of instances.

To study the four algorithms we present two tables (Tables 4 and 5): the first one is a comparison between TS and 3TS-BB and the second one between Walksat and 3Walksat-BB. Two criterions are used: the number of false clauses (fc.) and the number of steps (including local search moves and B&B backtracks) to obtain the best result (steps). For these two criterions we present the average (avg.) and its standard deviation (s.d.). Another interesting value is proposed in the last column: the percentage of improvement of the 3-version v.s the standard version, w.r.t. the average number of false clauses.

Families	Instances	sat03	var.	cl.
comb	comb1	419	5910	16804
	comb3	421	4774	16331
des-encryption	cnf-r4-b1-k1.1-comp	416	2424	14812
	cnf-r4-b1-k1.2-comp	417	2424	14812
ezfact-64	ezfact64_2	1518	3073	19785
	ezfact64_3	1519	3073	19785
	ezfact64_6	1522	3073	19785
	ezfact64_7	1523	3073	19785
kukula	am_6_6	362	2269	7814
	am_7_7	363	4264	14751
markstrom	mm-1x10-10-10-s.1	1488	1120	7220
	mm-2x2-5-5-s.1	1497	324	2064
okgen	okgen-c1935-v450-s569787048	1704	450	1935
	okgen-c2025-v450-s1380514806	1705	450	2025
Parity-32	par32-5-c	1539	1339	5350
	par32-5	1540	3176	10325
purdum	10142772393204023fw	1661	9366	37225
	10142772393204023nc	1662	8372	33248
	10142772393204023nw	1663	8589	34117
pyhala-braun	pyhala-braun-unsat-35-4-03	1543	7383	24320
	pyhala-braun-unsat-35-4-04	1544	7383	24320
	pyhala-braun-unsat-40-4-02	1546	9638	31795
	pyhala-braun-unsat-40-4-03	1547	9638	31795
qwh	qwh.40.544	1653	2843	22558
	qwh.40.560	1654	3100	26345
unif	unif-r4.25-v500-c2125-02-S1567634734	1111	500	2125
	unif-r4.25-v500-c2125-03-S948115330	1112	500	2125

Table 3: Instances selected from the SAT2003 competition. (sat03 → SAT2003 instance number, var. → variables, cl. → clauses)

Tables 4 and 5 show that the 3-valued framework permits to boost classic local search methods like TS and Walksat. Indeed, 3TS-BB provides an improvement which can be sometimes weak and exceptionally negative (−1.92% for ezfact64_3) but often really more significant for most of the instances (+93.71% for par32-5) and all the families. For Walksat, the improvement is less important with the 3-valued framework. However, 3Walksat-BB provides interesting results because 8 families are improved (4 more than 15%) and only 2 families are deteriorated (never more than -5%).

It is interesting to remark that the 3TS-BB and 3Walksat-BB results are better while the search power represented by the number of steps is in the same order of magnitude. Execution time is not mentioned in the tables but remains approximately the same with and without the 3-valued framework.

5 Discussion and Conclusion

Our 3-valued framework allows us to handle partial assignments and, thanks to these partial assignments, some characteristics of the formula, like the backbone [Dubois and Dequen, 2001], can be easily defined. The backbone of a satisfiable formula ϕ is the set of entailed literals. A literal l is entailed by ϕ if and only if $\phi \wedge (\neg l)$ is unsatisfiable. In the 3-valued framework, the backbone is the element $A \in Sol_\phi$ such that $\nexists A' \sqsupset A, A' \in Sol_\phi$.

In this paper, we have proposed a new resolution framework for SAT and MAX-SAT problems which includes a third truth value *undefined* in order to handle both partial and complete assignments. We have precisely studied the extension and behavior of local search algorithms using this third truth value. Two possible logical interpretations have been considered and it appears that the pessimistic approach provides better results than the optimistic approach.

We have then introduced this 3-valued framework inside Tabu Search and Walksat algorithms and the results are very

promising. Our future works will consist in developing new specific heuristics using this 3-valued framework and in extending other well known algorithms in a 3-valued version. This framework could also serve as basis for full hybridizations of complete and incomplete resolution techniques.

Acknowledgments

The work presented in this paper is partially supported by the CPER COM program. We would like to thank the anonymous referees for their useful comments.

References

- [Borchers and Furman, 1999] Brian Borchers and Judith Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2:299–306, 1999.
- [Davis *et al.*, 1962] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Com. of the ACM*, 5(7):394–397, 1962.
- [Dubois and Dequen, 2001] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proc. of IJCAI’01*, pages 248–253, M. Kaufmann, 2001.
- [Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. Freeman & Co, 1979.
- [Glover and Laguna, 1997] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.
- [Habet *et al.*, 2002] Djamel Habet, Chu Min Li, Laure Devendeville, and Michel Vasquez. A Hybrid Approach for SAT. In *Proc. of CP’02, LNCS*, pages 172–184. Springer, vol. 2470, 2002.
- [Hirsch and Kojevnikov, 2001] Edward A. Hirsch and Arist Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. Steklov Institute of Mathematics at St.Petersburg, 2001.
- [Mazure *et al.*, 1997] Bertrand Mazure, Lakhdar Sais, and Eric Grégoire. Tabu search for SAT. In *Proc. of AAAI-97/IAAI-97*, pages 281–285, AAAI Press, 1997.
- [Mazure *et al.*, 1998] Bertrand Mazure, Lakhdar Sais, and Eric Grégoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22:319–331, 1998.
- [Prestwich, 2000] Steve Prestwich. A hybrid search architecture applied to hard random 3-sat and low-autocorrelation binary sequences. In *Proc. of CP’00, LNCS*, pages 337–352. Springer, vol. 1894, 2000.
- [Prestwich, 2004] Steve Prestwich. Exploiting relaxation in local search. In *Proc. of the 1st Int. Workshop on Local Search Techniques in Constraint Satisfaction*, 2004.
- [Selman *et al.*, 1992] Bart Selman, Hector J. Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In *Proc. of AAAI’92*, pages 440–446, AAAI Press, 1992.

Families	sat03	3TS-BB				TS				Imp.	
		fc.		steps($\times 10^3$)		fc.		steps($\times 10^3$)		fc. %	
		avg.	s.d.	avg.	s.d.	avg.	s.d.	avg.	s.d.		
purdom	1661	304.00	4.00	665.05	23.72	497.50	16.50	273.81	172.52	+38.89	+33.39
	1662	256.33	9.09	462.10	300.75	350.00	3.00	401.52	259.26	+26.76	
	1663	247.50	1.50	663.28	122.31	378.00	10.00	669.85	60.97	+34.52	
kukula	362	22.58	2.36	346.57	291.14	48.00	8.72	803.18	225.66	+52.96	+52.88
	363	53.33	2.93	353.74	282.04	113.00	16.20	720.32	178.27	+52.81	
	des-encryption	416	35.75	2.83	372.87	197.30	53.33	2.29	1.32	0.27	
	417	36.42	2.38	202.87	258.25	50.67	4.38	66.50	216.62	+28.12	+30.54
comb	419	123.90	5.79	143.10	145.53	160.20	8.70	328.83	233.63	+22.66	+25.41
	421	79.33	5.01	243.34	190.96	110.42	4.83	156.43	87.09	+28.16	
	ezfact-64	1518	105.00	3.37	463.88	299.16	104.58	18.72	631.44	333.80	
	1519	102.00	4.47	388.21	209.20	100.08	11.21	669.38	315.35	-1.92	+1.05
	1522	108.83	5.26	383.12	319.05	112.33	10.92	672.35	249.31	+3.12	
	1523	110.00	4.64	488.94	287.57	113.42	15.67	623.18	276.93	+3.02	
markstrom	1488	9.92	0.82	382.26	255.69	16.08	5.12	304.65	303.09	+38.31	+29.93
	1497	6.08	0.78	359.18	266.21	7.75	1.59	102.11	198.37	+21.55	
	okgen	1704	4.33	1.04	306.77	249.03	4.58	1.20	340.69	283.10	
	1705	5.17	0.52	315.29	173.85	5.50	0.87	361.38	250.97	+6.00	+5.73
Parity-32	1539	9.67	0.81	488.20	290.15	65.58	1.23	173.14	213.85	+85.25	+89.48
	1540	9.75	1.09	509.59	225.89	155.00	1.68	192.88	262.24	+93.71	
	pyhala-braun	1543	239.00	3.00	92.11	20.37	377.00	0.00	269.85	30.10	
	1544	197.00	3.74	355.73	304.16	347.00	3.00	617.31	373.33	+43.23	+34.17
	1546	342.50	10.50	25.80	4.98	464.00	0.00	571.45	0.03	+26.19	
	1547	330.00	4.00	53.95	7.58	476.00	0.00	294.21	0.04	+30.67	
qwh	1653	125.67	9.67	52.49	63.89	130.08	10.03	33.86	60.82	+3.39	+0.95
	1654	132.00	9.05	56.67	69.93	130.08	10.84	48.06	119.94	-1.48	
	unif	1111	2.75	0.44	283.00	310.58	3.58	1.26	289.46	343.48	
	1112	1.75	0.60	379.46	353.53	5.42	1.43	135.37	81.25	+67.71	

Table 4: Comparisons between TS and 3TS-BB.

Instances	sat03	3Walksat-BB				Walksat				Imp. fc. %	
		fc.		steps($\times 10^3$)		fc.		steps($\times 10^3$)			
		avg.	s.d.	avg.	s.d.	avg.	s.d.	avg.	s.d.		
purdom	1661	372.45	12.65	635.00	270.69	412.05	12.77	487.16	248.84	+9.61	+21.61
	1662	327.85	14.54	455.00	257.83	374.55	8.45	508.58	295.37	+12.47	
	1663	353.00	15.70	540.00	281.78	393.70	9.07	656.23	247.86	+10.34	
kukula	362	1.10	0.44	333.41	265.67	1.80	2.38	409.08	263.57	+38.89	+23.42
	363	71.80	14.85	519.66	292.22	78.00	14.95	555.39	304.03	+7.95	
des-encryption	416	24.55	2.42	337.15	288.45	23.70	1.76	419.57	257.28	−3.59	−4.87
	417	23.95	1.66	551.11	322.33	23.35	1.68	458.96	295.70	−2.57	
comb	419	69.35	5.96	548.77	303.13	70.45	7.07	569.64	273.53	+1.56	+0.33
	421	39.60	4.09	452.94	309.93	39.25	5.62	527.25	338.57	−0.89	
ezfact-64	1518	70.95	3.92	544.84	279.54	86.95	4.69	393.75	280.57	+18.40	+18.43
	1519	72.70	4.56	629.37	294.40	90.40	3.57	520.31	276.70	+19.58	
	1522	73.95	4.24	518.53	284.14	88.60	2.62	308.05	267.34	+16.53	
	1523	72.55	3.93	548.05	280.23	89.80	3.67	587.71	287.36	+19.21	
markstrom	1488	10.70	1.49	570.15	239.46	11.45	1.63	400.54	269.03	+6.55	+2.75
	1497	4.85	0.57	423.88	302.64	4.80	0.51	214.27	240.11	−1.04	
okgen	1704	3.00	0.00	51.10	31.28	3.00	0.00	48.65	36.04	0.00	0
	1705	4.00	0.00	147.20	140.04	4.00	0.00	191.90	168.66	0.00	
Parity-32	1539	13.80	1.89	500.11	276.64	13.85	1.68	486.16	278.46	+0.36	+0.1
	1540	15.10	2.07	349.38	267.51	15.05	2.18	493.77	284.38	−0.33	
pyhala-braun	1543	175.85	7.27	452.92	302.98	216.70	11.33	505.76	300.51	+18.85	+19.48
	1544	172.10	7.63	555.63	293.43	215.90	10.19	542.80	313.90	+20.29	
	1546	238.90	9.32	660.00	259.62	293.25	9.15	551.42	280.12	+18.53	
	1547	239.70	10.85	625.00	289.61	300.55	8.69	552.30	263.27	+20.25	
qwh	1653	138.40	4.42	509.55	267.21	138.75	3.24	402.36	239.36	+0.25	+0.63
	1654	135.70	4.75	527.37	281.81	137.10	4.58	535.02	270.60	+1.02	
unif	1111	1.55	0.50	240.75	292.05	1.45	0.50	252.81	255.21	−6.90	−3.45
	1112	1.00	0.00	21.61	19.49	1.00	0.00	21.20	17.01	0.00	

Table 5: Comparisons between Walksat and 3Walksat-BB.

[Selman *et al.*, 1994] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proc. of AAAI'94*, pages 337–343, AAAI Press, 1994.

[Simon *et al.*, 2003] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The SAT2003 competition. Technical report, Sixth International Conference on the Theory and Applications of Satisfiability Testing, 2003.

[Spears, 1996] William M. Spears. Simulated annealing for hard satisfiability problems. In *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 533–558, vol. 26, 1996.

Resolution in Max-SAT and its relation to local consistency in weighted CSPs

Javier Larrosa, Federico Heras

{larrosa, fheras}@lsi.upc.edu

Dept. LSI, Universitat Politècnica de Catalunya
Barcelona, Spain

Abstract

Max-SAT is an optimization version of the well-known SAT problem. It is of great importance from both a theoretical and a practical point of view. In recent years, there has been considerable interest in finding efficient solving techniques [Alsinet *et al.*, 2003; Xing and Zhang, 2004; Shen and Zhang, 2004; de Givry *et al.*, 2003]. Most of this work focus on the computation of good quality lower bounds to be used within a branch and bound algorithm. Unfortunately, lower bounds are described in a procedural way. Because of that, it is difficult to realize the *logic* that is behind.

In this paper we introduce a logical framework for Max-SAT solving. Using this framework, we introduce an extension of the Davis-Putnam algorithm (that we call Max-DPLL) and the resolution rule. Our framework has the advantage of nicely integrating branch and bound concepts such as the lower and upper bound, as well as hiding away implementation details. We show that Max-DPLL augmented with a restricted form of resolution at each branching point is an effective solving strategy. We also show that the resulting algorithm is closely related with some local consistency properties developed for weighted constraint satisfaction problems.

1 Introduction

Since the eighties, both *boolean satisfiability* and *constraint satisfaction* have been the topic of intense algorithmic research. In both areas, the goal is to assign values to variables in such a way that no forbidden combination of values appear in the solution. In satisfiability, forbidden combinations are specified by means of *clauses*. In constraint satisfaction they are specified by means of arbitrary *constraints*. Given its similarity, it is hardly a surprise that both research communities have developed closely related techniques.

In both fields, the original decision problem (SAT and CSP, respectively) has been augmented to deal with unfeasible problems (namely, not all the clauses or constraints can be satisfied). The new goal is to find an assignment

that best respects the clauses/constraints. Two well-known examples are Max-SAT [Hansen and Jaumard, 1990] and *weighted* CSP (WCSP) [Bistarelli *et al.*, 1999], where most of recent algorithmic work has focused. In both cases the importance of the clauses/constraints is given by a *weight* and the goal is to minimize the sum of weights associated with the clauses/constraints violated by the assignment. It is known that Max-SAT instances can be translated into WCSP instances and vice versa [de Givry *et al.*, 2003]. In both cases the main solving technique are enumeration algorithms based on branch and bound search.

In the CSP side, several local consistency properties have been recently generalized to the WCSP framework [Cooper, 2003; Larrosa and Schiex, 2004; 2003; de Givry *et al.*, 2005]. As a result, a new family of algorithms have been proposed. Besides being efficient, these algorithms have a highly desirable property: they can be neatly described as a basic backtracking search in which certain local consistency property is enforced at every search state.

In [de Givry *et al.*, 2003], it was shown that Max-SAT instances could be efficiently solved by translating them into equivalent WCSP instances and later using a WCSP solver. The level of local consistency maintained by the WCSP solver was fundamental in the efficiency of the algorithm.

In this paper we analyze the interpretation of WCSP local consistency properties and related algorithms when applied to Max-SAT instances. To facilitate the connection, we start by providing a reformulation of Max-SAT in which it is possible to explicitly express a lower and an upper bound of the optimal cost (Section 3). Such reformulation makes possible an elegant extension of DPLL in which each branching point is just a Max-SAT instance, similarly to what happens with DPLL in SAT (Section 4). In Section 5 we present our main contribution: a generalization of the resolution rule ($x \vee A, \bar{x} \vee B \Rightarrow A \vee B$) and the proof that neighborhood resolution (i.e., a restricted form of resolution) suffices to enforce (weighted) node and arc consistency. In Section 6 we put our approach in context with other Max-SAT algorithms. Finally, in Section 7 we provide conclusions and directions of future work.

2 Preliminaries

In the sequel $X = \{x_1, \dots, x_n\}$ will denote a set of boolean variables. They take values over the set $\{t, f\}$, which stands

for *true* and *false*, respectively. A *literal* l is either a variable x or its negation $\bar{x}$. $\bar{l}$ stands for the negation of l .

If variable x is instantiated to $\mathbf{t}$, noted $x \leftarrow \mathbf{t}$ literal x is satisfied and literal $\bar{x}$ is falsified. Similarly, if x is instantiated to $\mathbf{f}$, $\bar{x}$ is satisfied and x is falsified. An *assignment* is an instantiation of a subset of the variables. The assignment is *complete* if it instantiates all the variables in X (otherwise it is partial). An assignment satisfies a clause (i.e., a disjunction of literals) C iff it satisfies one or more of its literals. It satisfies a set of clauses $\mathcal{F}$ iff it satisfies all its clauses. A satisfying complete assignment is called a *model*. Given a boolean formula encoded by a set of clauses $\mathcal{F}$, the SAT problem consists in determining whether there is any model for it or not.

We will use the symbol $\square$ to denote the *empty clause* which, obviously, cannot be satisfied. When $\square \in \mathcal{F}$ we say that there is an explicit *contradiction*.

When there is no model for the formula $\mathcal{F}$, one may be interested in finding the complete assignment with minimum number of violated clauses. This problem is known as Max-SAT.

3 An equivalent reformulation of Max-SAT

There is a *weighted* version of Max-SAT in which (weighted) clauses are pairs (C, w) such that C is a classical clause and w is the cost of its falsification. In weighted Max-SAT $\mathcal{F}$ is a set of weighted clauses. The cost of an assignment is the sum of weights of all the clauses that it falsifies. The goal is to find complete assignments with minimal cost. We make the usual assumption of weights being natural numbers.

It is easy to see that Max-SAT and weighted Max-SAT have exactly the same expressive power. A Max-SAT instance can be rewritten as a weighted instance replacing every clause C by a weighted clause $(C, 1)$. A weighted Max-SAT instance can be rewritten as a Max-SAT instance replacing every weighted clause (C, w) by w copies of clause C . Clearly, weighted Max-SAT encodings may be exponentially more compact than Max-SAT. Thus, in the following, we will assume, without loss of generality, weighted Max-SAT.

Following previous work in weighted constraint satisfaction [Larrosa and Schiex, 2004], we assume the existence of a known upper bound k of the optimal solution. This is also done without loss of generality because, if a tight upper bound is not known, k can be set to the sum of weights of all the clauses.

Consider the set $\mathcal{F}$ of weighed clauses. We say that a *model* is a *complete assignment with cost less than k* . Max-SAT is the problem of *finding a model of minimal cost*, if there is any.

Observe that weights $w \geq k$ indicate that the associated clause *must be necessarily satisfied*. Thus, we can replace every weight $w \geq k$ by k without changing the problem. Thus, without loss of generality we assume all costs in the interval $[0..k]$ and, accordingly, redefine the sum of costs as

$$a \oplus b = \min\{a + b, k\}$$

in order to keep the result within the interval. For convenience of notation, we will refer to k as $\top$. We say that a weighted clause is *hard* (or *mandatory*) iff its weight is $\top$. Observe that Max-SAT with $\top = 1$ is equivalent to SAT.

Example 1 Consider the formula $\{(x, 1), (\bar{y}, 5), (\bar{x} \vee y, 2), (x \vee y, 4)\}$, with $\top = 5$. The second clause is *hard*. The assignment $x = \mathbf{f}, y = \mathbf{f}$ is not a model because its cost is $1 \oplus 4 = \top$. The assignment $x \leftarrow \mathbf{t}, y \leftarrow \mathbf{f}$ is a (optimal) model with cost 2.

Note that in Max-SAT truth tables are tables with a cost associated to each truth assignments. A brute-force solving method consists in computing the truth table of the input formula and finding the minimal cost model. For instance, the truth-table of the previous formula is,

$x y$	cost
$\mathbf{f f}$	$1 \oplus 4 = \top$
$\mathbf{f t}$	$1 \oplus \top = \top$
$\mathbf{t f}$	2
$\mathbf{t t}$	$\top$

It is worth mentioning the role of the empty clause $(\square, w)$. Since it cannot be satisfied, w will be added to the cost of any model. Therefore, w is an explicit *lower bound* of the optimal model. When the lower bound and the upper bound have the same value (i.e., $(\square, \top) \in \mathcal{F}$) the formula does not have any model and we call this situation an explicit *contradiction*.

4 Generalization of DPLL to Max-SAT

4.1 Max-SAT Basic Simplification Rules

SAT solvers take advantage from some equivalence rules that are used to simplify the CNF formula without changing its set of models. Not all of these formulas can be applied directly to Max-SAT. In this Section we state some useful Max-SAT specific rules. We use the notation $[P, \dots, Q] \Rightarrow [R, \dots, S]$, where $P, Q, \dots$ are weighted clauses. It means that if there are in $\mathcal{F}$ weighted clauses matching with $[P, \dots, Q]$, they can be replaced by $[R, \dots, S]$. A and B are arbitrary disjunctions of literals.

- BR1: $[(A, \top), (A \vee B, w)] \Rightarrow [(A, \top)]$
- BR2: $[(A, w), (A, u)] \Rightarrow [(A, w \oplus u)]$
- BR3: If $(w \oplus u = \top)$ then $[(A, w), (A \vee B, u)] \Rightarrow [(A, w), (A \vee B, \top)]$
- BR4: $[(A, 0)] \Rightarrow []$

BR1 shows that classical absorption can only be applied when the absorbing clause is hard. BR2 generalizes the standard idempotency of the conjunction: In Max-SAT the weights of the repeated clauses must be added in the resulting clause. BR3 is used to harden a soft clause. BR4 indicates that cost-free clause can be eliminated. The correctness of these equivalences is direct and we omit the proof.

Example 2 Consider the formula $\{(x, 1), (x, 1), (x \vee y, 3), (x \vee y \vee z, 1)\}$, with $\top = 5$. We can apply BR2, which produces $\{(x, 2), (x \vee y, 3), (x \vee y \vee z, 1)\}$. Now, we can apply BR3, producing $\{(x, 2), (x \vee y, \top), (x \vee y \vee z, 1)\}$. Finally, BR1 produces $\{(x, 2), (x \vee y, \top)\}$. The equivalence between the original and the final formula can be checked by constructing and comparing the costs of the two truth-tables.

4.2 Max-DPLL

Davis Putnam (DPLL) is the most popular algorithm for SAT and the starting point of most state-of-the-art solvers [Davis *et al.*, 1962]. It takes as input a CNF formula $\mathcal{F}$ and decides whether or not there exists a model. In this Section, we present a natural extension of DPLL to Max-SAT that we call Max-DPLL. Let $\mathcal{F}$ be a weighted CNF formula and $\top$ its upper bound. $\text{Max-DPLL}(\mathcal{F}, \top)$ returns the cost of the optimal model of $\mathcal{F}$ if there is such a model, else it returns $\top$. The following description is inspired by the description of DPLL given in [Bacchus, 2002].

As in classical SAT, Max-DPLL performs basic simplifications on its input prior to invoking itself recursively. The *instantiation* of a variable by forcing the satisfaction of a literal l , denoted $\mathcal{F}[l]$ produces a new formula generated from $\mathcal{F}$ as follows: all clauses containing l are eliminated, and l is removed from all clauses where it appears. *Unit Clause Reduction* (UCR) is another simplification rule that selects a clause $(l, \top)$ (namely, a unit hard clause) and instantiates the corresponding variable in accordance to the literal in that clause. *Unit Propagation* (UP) is the algorithm that performs UCR and the basic simplification rules BR1 – 4 until either (a) a contradiction is achieved, or (b) there are no more possible simplifications to do.

Example 3 Consider the application of UP to the formula $\{(\square, 1), (\bar{x}, 4), (x, 1), (y, 3), (\bar{y}, 1), (\bar{x} \vee \bar{y} \vee z, 1)\}$, with $\top = 5$. Rule BR3 transforms it into $\{(\square, 1), (\bar{x}, \top), (x, 1), (y, 3), (\bar{y}, 1), (\bar{x} \vee \bar{y} \vee z, 1)\}$. UCR instantiates $\bar{x}$ and produces $\{(\square, 3), (y, 3), (\bar{y}, 1), (\bar{y} \vee z, 1)\}$. Again BR3 produces $\{(\square, 3), (y, \top), (\bar{y}, 1), (\bar{y} \vee z, 1)\}$, which allows UCR to instantiate y , $\{(\square, 4), (z, 1)\}$. Again BR3 produces $\{(\square, 4), (z, \top)\}$, which UCR to instantiate z producing the trivial $\{(\square, 4)\}$. The model of the original formula $(\bar{x}, y, z)$ can be constructed by tracking back the truth assignments made by UCR.

A recursive description of Max-DPLL is given in Algorithm 1. First, UP is applied to the input formula (line 1). If the resulting formula contains a contradiction, the algorithm returns $\top$ and backtracks (line 2). Else, if it does not contain any variable the trivial cost of the optimal model is returned (lines 3 and 4). Otherwise, an arbitrary literal l is selected (line 5). The formula is instantiated with l and $\bar{l}$ and Max-DPLL is recursively called with each case (lines 6 and 7). Observe that the first recursive call is made with the $\top$ inherited from its parent, but the second call uses the output of the first call. This implements the typical upper bound updating of branch and bound. Finally, the best value of the two recursive calls is returned (line 8). Note that if Max-DPLL is executed with a SAT instance (i.e., $\top = 1$) it behaves exactly as DPLL.

Figure 1 reports some empirical evaluation of Max-DPLL on 40-variable random instances of Max-2-SAT and Max-3-SAT generated with the *Cnfgen* generator.¹ The horizontal axis indicates the number of clauses and the vertical axis indicates the search effort as the number of visited nodes. Each

¹A. van Gelder [ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances](http://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances)

Function Max-DPLL($\mathcal{F}, \top$) : nat

```

1   $\mathcal{F} := \text{UP}(\mathcal{F});$ 
2  if  $(\square, \top) \in \mathcal{F}$  then return  $\top$ ;
3  if  $\mathcal{F} = \emptyset$  then return 0;
4  if  $\mathcal{F} = \{(\square, w)\}$  then return  $w$ ;
5   $l := \text{SelectLit}(\mathcal{F});$ 
6   $v := \text{Max-DPLL}(\mathcal{F}[l], \top);$ 
7   $v := \text{Max-DPLL}(\mathcal{F}[\bar{l}], v);$ 
8  return  $v$ ;
```

Algorithm 1: Max-DPLL. $\mathcal{F}$ is a set of weighted clauses with all weights in the interval $[0, \dots, \top]$. If the weighted formula has models, Max-DPLL returns the cost of the optimal one, else it returns $\top$.

point is the mean over 10 instances. The results of Max-DPLL are those labelled as UP (ignore, for the moment, the other curves). As can be observe, the performance of the algorithm degenerates as the number of clauses increases.

Max-DPLL can be enhanced by *dominance* rules that exploit situations where it is easy to detect that one literal is never worse than its negation. A well-known SAT case, also applicable to Max-SAT, is the *pure literal rule*. It says that if there is a literal such that it appears in the formula and its negation does not appear, then all clauses mentioning it can be removed. More sophisticated dominance rules for Max-SAT and WCSP can be found in [Xing and Zhang, 2004; de Givry, 2004].

5 Resolution for Max-SAT

While DPLL seems to be the best option to find models, *resolution* might be more appropriate to detect contradictions. [Robinson, 1965] showed that the resolution rule is sound and complete for SAT, although it is usually too space consuming. In the SAT context, the performance of DPLL has been improved by the addition of limited forms of resolution at each search node [Gelder, 1995; Rish and Dechter, 2000; Bacchus, 2002; Drake *et al.*, 2002] in order to anticipate the detection of dead-ends. In this section we generalize the resolution rule to Max-SAT. Then we show that some local consistency techniques used in [Larrosa and Schiex, 2004] are just the application of a restricted form of the weighted resolution rule.

First we define the *subtraction* of weights ($\ominus$): Let $a, b \in [0, \dots, \top]$ be two weights such that $a \geq b$,

$$a \ominus b = \begin{cases} a - b & : a \neq \top \\ \top & : a = \top \end{cases}$$

The *weighted resolution* rule (RES) is defined as,

$$(x \vee A, u), (\bar{x} \vee B, w) \Rightarrow \begin{cases} (A \vee B, m) \\ (x \vee A, u \ominus m) \\ (\bar{x} \vee B, w \ominus m) \\ (x \vee A \vee B, m) \\ (\bar{x} \vee A \vee B, m) \end{cases}$$

where A and B are arbitrary disjunctions of literals and $m = \min\{u, w\}$. Variable x is called the clashing variable. Observe that in the $u = w = \top$ case the fourth and fifth new

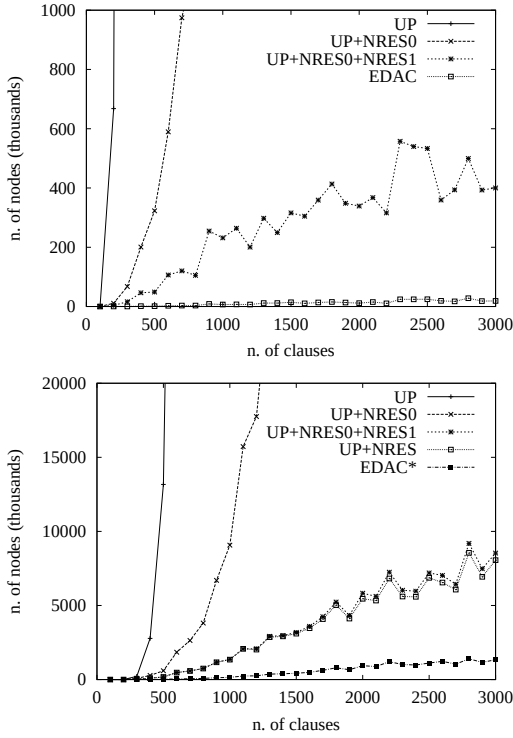


Figure 1: Experimental results on random Max-2SAT (top) and Max-3SAT (bottom).

clauses can be removed (by BR1), yielding a rule similar to classical resolution. Observe as well that in the $u = w < \top$ case, the second and third new clauses will have weight 0 and can be removed (by BR4). In the $u \neq w$ case, either the second or the third new clause will have weight 0 and can be removed (by BR4). Recall that in RES, the left-hand clauses are replaced by the right-hand, while in classical resolution right-hand clauses are just added.

Theorem 1 *The transformation rule RES is sound. Namely, it preserves the set of models and their cost.*

proof 1 *The following table contains in the first columns all the truth assignments, in the second column the cost of the assignment according to the clauses on the left-hand of the RES rule, and in the third column the cost of the assignment according to the clauses on the right-hand of the RES rule. As it can be observed, the costs are the same, so the resulting problem is equivalent.*

$x \ A \ B$	Left	Right
f f f	u	$m \oplus (u \oplus m)$
f f t	u	$m \oplus (u \oplus m)$
f t f	0	0
f t t	0	0
t f f	w	$m \oplus (w \oplus m)$
t f t	0	0
t t f	w	$m \oplus (w \oplus m)$
t t t	0	0

The interest of RES is that it makes explicit some previously hidden connection between variables in A and B . It is

well known that in the SAT case (i.e., $\top = 1$) the rule is complete (its application suffices to achieve a contradiction). We are still investigating under which other cases the completeness of RES is guaranteed. One important problem we have detected is that the fourth and fifth new weighted formulas (i.e., $(x \vee A \vee B, m)$, $(\bar{x} \vee A \vee B, m)$) may not be in conjunctive normal form.

5.1 Neighborhood Resolution

Neighborhood Resolution [Cha and Iwama, 1996] is the classical resolution rule restricted to pairs of clauses that differ only in the clashing variable. Similarly, in the Max-SAT context we define the neighborhood resolution rule (NRES) as RES restricted to the $A = B$ case, which simplifies to,

$$(x \vee A, u), (\bar{x} \vee A, w) \Rightarrow \begin{cases} (A, m) \\ (x \vee A, u \oplus m) \\ (\bar{x} \vee A, w \oplus m) \end{cases}$$

with $m = \min\{u, w\}$. This rule is specially useful because it projects to A costs that were implicit in the formula. We demonstrate the interest of NRES considering its application to bounded-size clauses. Let $NRES_k$ denote NRES restricted to $|A| = k$. $NRES_0$ yields,

$$(x, u), (\bar{x}, w) \Rightarrow \begin{cases} (\square, m) \\ (x, u \oplus m) \\ (\bar{x}, w \oplus m) \end{cases}$$

This rule is extremely useful because it produces a direct increment of the lower bound, which may raise a contradiction, or produce new unit clause reductions.

Example 4 *Consider the formula $\{(x, 1), (\bar{x}, 2), (y, 1), (\bar{y}, 1), (z, 1)\}$ with $\top = 3$. UP is unable to simplify the problem. Nevertheless, $NRES_0$ can be applied to the first and second clauses producing $\{(\square, 1), (\bar{x}, 1), (y, 1), (\bar{y}, 1), (z, 1)\}$. Applying $NRES_0$ to the third and fourth clauses produces $\{(\square, 2), (\bar{x}, 1), (z, 1)\}$. Using BR3 we obtain $\{(\square, 2), (\bar{x}, \top), (z, \top)\}$. The two unit clauses can be reduced, producing $\{(\square, 2)\}$.*

It is interesting to observe that the application of $NRES_0$ is somehow similar to the computation of the lower bound of [Alsinet et al., 2003]. The practical importance of adding $NRES_0$ to Max-DPLL is illustrated in Figure 1. The lines labelled UP+NRES0 report the efficiency of Max-DPLL when UP is augmented with $NRES_0$ until quiescence. As it can be observed $NRES_0$ produces huge savings over the very inefficient Max-DPLL.

The rule $NRES_1$ is,

$$(x \vee l, u), (\bar{x} \vee l, w) \Rightarrow \begin{cases} (l, m) \\ (x \vee l, u \oplus m) \\ (\bar{x} \vee l, w \oplus m) \end{cases}$$

This rule is also of great interest because it increases the weight of a unary clauses, which may allow further application $NRES_0$ and UCR.

Example 5 *Consider the formula $\{(x \vee y, 1), (\bar{x} \vee y, 1), (\bar{y}, 1), (z, 1)\}$ with $\top = 2$. Neither UP nor $NRES_0$ can simplify the problem. However, $NRES_1$ produces*

$\{(\bar{y}, 1), (y, 1), (z, 1)\}$ which allows $NRES_0$ to transform the problem into $\{(\square, 1), (z, 1)\}$. We can apply BR3 to the unary clause obtaining $\{(\square, 1), (z, \top)\}$. Now, UCR produces $\{(\square, 1)\}$.

The practical importance of adding $NRES_1$ to Max-DPLL is also illustrated in Figure 1. The lines labelled UP+NRES0+NRES1 report the efficiency of Max-DPLL when UP is augmented with $NRES_0$ and $NRES_1$ until quiescence. As it can be observed, the addition of $NRES_1$ also produces huge gains. For the sake of completeness, we also evaluate the effect of $NRES_k$ for $k > 1$. Obviously, in 2-SAT it does not have any effect, since the application of $NRES_k$ requires clauses of length $k + 1$. In 3-SAT, $NRES_2$ can only be applied in the original ternary clauses. Its effect is reported in Figure 1 under the label UP+NRES. It can be observed that $NRES_2$ has a very limited effect.

5.2 Neighborhood Resolution and Local Consistency

In this Section we relate the simplification rules discussed along the paper with local consistency properties developed for WCSP. In order to do so, we recall that the usual way to map a Max-SAT instance with a WCSP is to group clauses mentioning exactly the same set of variables and associate them to a cost function f defined as follows: Let $\mathcal{V} \subset \mathcal{F}$ be the group of clauses over the set of variables $Y \subset X$. $\mathcal{V}$ defines a cost function f with scope Y . Let t be an instantiation of the variables in Y . If t falsifies a clause $(C, w) \in \mathcal{V}$, $f(t) = w$, else $f(t) = 0$. Now it is straightforward to redefine the WCSP local consistency properties in Max-SAT terms. In the following consider a boolean formula $\mathcal{F}$ where $w(\square)$ denotes the weight of the empty clause; $w(i)$ and $w(\bar{i})$ the weights of the unary clauses x_i and $\bar{x}_i$, respectively; analogously $w(ij)$, $w(\bar{i}\bar{j})$, $w(i\bar{j})$ and $w(\bar{i}j)$ are the weights of the four possible binary clauses over x_i and x_j . If any of these clauses is not in $\mathcal{F}$ the corresponding weight is 0.

Definition 1 $\mathcal{F}$ is *node-consistent (NC)* iff for all variable x_i , $w(\square) \oplus w(i) < \top$ and $w(\square) \oplus w(\bar{i}) < \top$.

Theorem 2 Algorithm UP enforces the NC property.

proof 2 Suppose that $\mathcal{F}$ is not NC. Then there is some clause $(l, w) \in \mathcal{F}$ such that $w(\square) \oplus w = \top$. Therefore BR3 can be applied replacing clause (l, w) by $(l, \top)$. It will allow the application of UCR which will eliminate the clause.

Definition 2 $\mathcal{F}$ is *star node-consistent (NC*)* iff it is NC and for all variable x_i , $w(i) = 0$ or $w(\bar{i}) = 0$

Theorem 3 Let UP +NRES₀ denote the algorithm that applies UP and NRES₀ until quiescence. It enforces NC*.

proof 3 We only need to proof that the application of NRES₀ guarantees that for all variable x_i , $w(i) = 0$ or $w(\bar{i}) = 0$. Assume that $0 < w(i) \leq w(\bar{i})$. There are two possible situations. The first one is $w(i) = \top$. In that case UP will trigger UCR and x_i will disappear from the formula. The second situation is $w(i) < \top$. Then, the application of NRES₀ will add $w(i)$ to $w(\square)$ and replace $(x_i, w(i)), (\bar{x}_i, w(\bar{i}))$ by $(x_i, 0), (\bar{x}_i, w(i) \ominus w(i))$.

The algorithm to enforce NC* introduced in [Larrosa and Schiex, 2004] proves the following result,

Corollary 1 UP +NRES₀ can be implemented with time complexity $O(n)$, where n is the number of variables in the formula.

Definition 3 $\mathcal{F}$ is *arc-consistent (AC*)* iff it is NC* and for all pair of variables (x_i, x_j) , $\min\{w(ij), w(\bar{i}\bar{j})\} = 0$ and $\min\{w(i\bar{j}), w(\bar{i}j)\} = 0$

Theorem 4 Let UP +NRES₀ +NRES₁ denote the algorithm that applies UP, NRES₀ and NRES₁ until quiescence. It enforces arc-consistency (AC*).

proof 4 We only need to proof that the application of NRES₁ guarantees that for all pair of variables (x_i, x_j) , $\min\{w(ij), w(\bar{i}\bar{j})\} = 0$ and $\min\{w(i\bar{j}), w(\bar{i}j)\} = 0$. We proof the first condition (the second is similar). Assume that $0 < w(ij) \leq w(\bar{i}\bar{j})$. There are two possible situations. The first one is that $w(ij) = \top$ (which implies $w(\bar{i}\bar{j}) = \top$). In that case NRES₁ will add $(x_i, \top)$ to the formula. Then, UCR will be triggered and x_i will disappear from the formula. The second situation is $w(ij) < \top$. Then, the application of NRES₁ will add $w(ij)$ to $w(i)$ and replace $(x_i \vee x_j, w(ij)), (x_i \vee \bar{x}_j, w(\bar{i}j))$ by $(x_i \vee x_j, 0), (x_i \vee \bar{x}_j, w(ij) \ominus w(ij))$.

The algorithm to enforce AC* introduced in [Larrosa and Schiex, 2004] proofs the following result.

Corollary 2 UP +NRES₀ +NRES₁ can be implemented with time complexity $O(n^2)$, where n is the number of variables in the formula.

6 Neighborhood Resolution and state-of-the-art Max-SAT lower bounds

In the last years several algorithms for Max-SAT have been proposed [Alsinet et al., 2003; Xing and Zhang, 2004; Shen and Zhang, 2004]. All these works have in common a basic branch and bound algorithm. They mainly differ in the lower bound that they use. [Shen and Zhang, 2004] show that their lower bound is better (i.e., higher than or equal) than [Alsinet et al., 2003] and, under some reasonable conditions, it is also better than [Xing and Zhang, 2004]. In the following, by means of two examples, we show that UP +NRES₀ +NRES₁ is not comparable with them.

Consider the formula $\{(x \vee z, 1), (\bar{x} \vee z, 1), (y \vee \bar{z}, 1), (\bar{y} \vee \bar{z}, 1)\}$. Both [Xing and Zhang, 2004] and [Shen and Zhang, 2004] would compute a lower bound 0. However, NRES₁ can be applied twice, producing the equivalent formula $\{(z, 1), (\bar{z}, 1)\}$ and now NRES₀ would transform the problem into $\{(\square, 1)\}$, which means a lower bound of 1.

Consider now the formula $\{(x, 1), (y, 1), (\bar{x} \vee \bar{y}, 1)\}$. While both [Xing and Zhang, 2004] and [Shen and Zhang, 2004] would compute a lower bound of 1, NRES cannot be applied, so the implicit lower bound of our algorithm would be 0. Nevertheless, it is important to observe that, if we encode this problem as a weighted CSP and enforce a stronger form of consistency called existential directional arc consistency (EDAC*) [de Givry et al., 2005], we transform the formula into the equivalent $\{(\square, 1), (x \vee y, 1)\}$ which also has

an implicit lower bound of 1. The performance of enforcing EDAC* on random Max-SAT instances is also illustrated in Figure 1. As it can be seen, it clearly provides further improvement over the application of the NRES rule. Experiments on Max-SAT using weighted CSP technology can be performed using the freely available *toolbar* solver.²

7 Conclusions and Future work

Motivated by the success of [de Givry *et al.*, 2003] in solving Max-SAT instances as weighted CSPs, we have studied the interpretation of WCSP local consistency properties within the Max-SAT context. The result of our work is a logical framework for Max-SAT in which the solving process can be seen as a set of transformation rules. Interestingly, our approach leads to a natural extension of the Davis-Putnam algorithm as well as an extension of the resolution rule (RES). The application of a limited form of RES, called neighborhood resolution (NRES), provides an interesting and effective algorithm. We have shown the relation between the application of NRES and some local consistency properties in weighted CSP. Finally, we have also shown that NRES provides (implicit) lower bounds not subsumed by state-of-the-art Max-SAT solvers.

We believe our work leaves several directions of future work. First, it would be interesting to analyze under which conditions the application of RES provides a complete method for Max-SAT. Since it has been shown that *directional resolution* is just an instantiation of *adaptive consistency* [Rish and Dechter, 2000], it would also be interesting to know if there is any relation between the application of RES and the more general *bucket elimination* [Dechter, 1999].

Our framework facilitates the relation between Max-SAT and weighted CSP solving. The resolution rule that we have introduced seems only valid to enforce node and arc consistency. However, it is known in the WCSP context that stronger forms of local consistency such as EDAC* may be more effective in practice [de Givry *et al.*, 2005]. Thus, it seems natural to search for new transformation rules such that, when added to NRES, can achieve higher levels of local consistency.

References

- [Alsinet *et al.*, 2003] T. Alsinet, F. Manya, and J. Planas. Improved branch and bound algorithms for max-sat. In *Proc. of the 6th SAT*, pages 408–415, 2003.
- [Bacchus, 2002] Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *Proceedings of the 18th AAAI*, pages 613–619, 2002.
- [Bistarelli *et al.*, 1999] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs: Frameworks, properties and comparison. *Constraints*, 4:199–240, 1999.
- [Cha and Iwama, 1996] Byungki Cha and Kazuo Iwama. Adding new clauses for faster local search. In *Proc. of the 13th AAAI*, pages 332–337, Portland, OR, 1996.
- [Cooper, 2003] M. Cooper. Reductions operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3):311–342, 2003.
- [Davis *et al.*, 1962] M. Davis, G. Logemann, and G. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
- [de Givry *et al.*, 2003] S. de Givry, J. Larrosa, P. Meseguer, and T. Schiex. Solving max-sat as weighted csp. In *Proc. of the 9th CP*, pages 363–376, Kinsale, Ireland, 2003. LNCS 2833. Springer Verlag.
- [de Givry *et al.*, 2005] S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted csp. In *Proc. of the 19th IJCAI*, Edinburgh, U.K., August 2005.
- [de Givry, 2004] S. de Givry. Singleton consistency and dominance testing for weighted csp. In *Proc. of the 6th Intl. Workshop on Soft Constraints and Preferences*, pages 363–376, Toronto, Canada, 2004.
- [Dechter, 1999] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.
- [Drake *et al.*, 2002] L. Drake, A. Frisch, and T. Walsh. Adding resolution to the dpll procedure for boolean satisfiability. In *Proceedings of 5th SAT*, pages 122–129, 2002.
- [Gelder, 1995] A. Van Gelder. *Satisfiability testing with more reasoning and less guessing*, pages 0–1. American Mathematical Society, 1995.
- [Hansen and Jaumard, 1990] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.
- [Larrosa and Schiex, 2003] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted csp. In *Proc. of the 18th IJCAI*, Acapulco, Mexico, August 2003.
- [Larrosa and Schiex, 2004] J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc-consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.
- [Rish and Dechter, 2000] I. Rish and R. Dechter. Resolution vs. sat: two approaches to sat. *Journal of Automated Reasoning*, 24(1):225–275, 2000.
- [Robinson, 1965] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
- [Shen and Zhang, 2004] Haiou Shen and Hantao Zhang. Study of lower bounds for max-2-sat. In *Proceedings of the 19th AAAI*, 2004.
- [Xing and Zhang, 2004] Zhao Xing and Weixiong Zhang. Efficient strategies for (weighted) maximum satisfiability. In *Proc. of the 10th CP*, pages 690–705, Toronto, Canada, 2004. LNCS 3258. Springer Verlag.

²<http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro>

A Greedy Approach to Establish Singleton Arc Consistency

Christophe Lecoutre and Stéphane Cardon

CRIL-CNRS FRE 2499,

Université d'Artois

Lens, France

{*lecoutre, cardon*}@cril.univ-artois.fr

Abstract

In this paper, we propose a new approach to establish Singleton Arc Consistency (SAC) on constraint networks. While the principle of existing SAC algorithms involves performing a breadth-first search up to a depth equal to 1, the principle of the two algorithms introduced in this paper involves performing several runs of a greedy search (where at each step, arc consistency is maintained). It is then an original illustration of applying inference (i.e. establishing singleton arc consistency) by search. Using a greedy search allows benefiting from the incrementality of arc consistency, learning relevant information from conflicts and, potentially finding solution(s) during the inference process. Furthermore, both space and time complexities are quite competitive.

1 Introduction

Inference and search are two categories of techniques for processing constraints [Dechter, 2003]. On the one hand, inference is used to transform a problem into an equivalent form which is either directly used to show the satisfiability or unsatisfiability of the problem, or simpler to be handled by a search algorithm. Inference aims at modifying a constraint network by employing structural methods such as variable elimination and tree clustering, or filtering methods based on properties such as arc consistency and path consistency. On the other hand, search is used to traverse the space delimited by the domains of all variables of the problem. Search can be systematic and complete by relying on breadth-first or depth-first exploration with backtracking, or stochastic and incomplete by relying on greedy exploration and randomized heuristics.

One of the most popular systematic search algorithms to solve instances of the Constraint Satisfaction Problem (CSP) is called MAC [Sabin and Freuder, 1994]. MAC interleaves inference and search since at each step of a depth-first exploration with backtracking, a local consistency called arc consistency is maintained. However, since the introduction of stronger consistencies such as max-restricted path consistency [Debruyne and Bessière, 1997a] and singleton arc consistency [Debruyne and Bessière, 1997b], one issue has been

the practical interest of utilizing such consistencies, instead of arc consistency, before or during search.

There is a recent focus about singleton consistencies, and more particularly about SAC (Singleton Arc Consistency), as illustrated by recent works of [Debruyne and Bessière, 1997b; Prosser *et al.*, 2000; Bartak, 2004; Bessière and Debruyne, 2004; 2005]. A constraint network is singleton arc consistent iff any singleton check does not show unsatisfiability, i.e., iff after performing any variable assignment, enforcing arc consistency on the resulting network does not entail a domain wipe-out.

In this paper, we propose two new algorithms, denoted SAC-3 and SAC-3+, to establish singleton arc consistency. While SAC algorithms introduced so far perform a breadth-first search up to a depth equal to 1, these two new algorithms perform several runs of a greedy search (where at each step, arc consistency is maintained). However, unlike SAC-3+, SAC-3 does not record the context of performed runs.

We have identified several advantages to adopt this approach:

- extra space requirement is limited,
- both algorithms benefit from the incrementality of arc consistency,
- using a greedy search enables learning relevant information from conflicts,
- it is possible to find solution(s) while establishing the consistency,
- time complexity of both algorithms is quite competitive.

More precisely, the good space complexity of both algorithms allows to use them on large constraint networks. In particular, SAC-3 admits the space complexity of the underlying arc consistency algorithm. Next, when a greedy search maintaining arc consistency is used, we naturally benefit from the incrementality of arc consistency, i.e., the fact that iteratively establishing arc consistency on a more and more reduced search space is less penalizing than repeatedly establishing it on the original search space. Besides, when a dead-end is encountered during a greedy search, a no-good can be recorded and/or the origin of the failure taken into account. Also, some solutions may be found by the algorithm. Finally, if the instance contains a large under-constrained part, a very efficient time complexity can be expected.

2 Preliminaries

A constraint network consists of a finite set of variables such that each variable X has an associated domain $\text{dom}(X)$ denoting the set of values allowed for X , and a finite set of constraints such that each constraint C has an associated relation $\text{rel}(C)$ denoting the set of tuples allowed for the variables $\text{vars}(C)$ involved in C . A solution to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied. A constraint network is said to be satisfiable if it admits at least a solution.

The Constraint Satisfaction Problem (CSP), whose task is to determine whether or not a given constraint network is satisfiable, is NP-complete. A constraint network is also called CSP instance. To solve a CSP instance, a depth-first search algorithm with backtracking can be applied, where at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation. Usually, constraint propagation algorithms, which are based on some constraint network properties such as arc consistency, remove some values which can not occur in any solution.

Definition 1 Let $P = (\mathcal{X}, \mathcal{C})$ be a constraint network, $C \in \mathcal{C}$, $X \in \text{vars}(C)$ and $a \in \text{dom}(X)$. (X, a) is said to be arc consistent wrt C iff there exists a support of (X, a) in C , i.e., a tuple $t \in \text{rel}(C)$ such that $t[X] = a$. P is said to be arc consistent iff $\forall X \in \mathcal{X}$, $\text{dom}(X) \neq \emptyset$ and $\forall C \in \mathcal{C}$, $\forall X \in \text{vars}(C)$, $\forall a \in \text{dom}(X)$, (X, a) is arc consistent wrt C .

$\text{AC}(P)$ will denote the constraint network obtained after enforcing Arc Consistency (AC) on a given constraint network P . $\text{AC}(P)$ is such that all values of P that are not arc consistent have been removed. Note that a value will usually refer to a pair (X, a) with $X \in \mathcal{X}$ and $a \in \text{dom}(X)$. If there is a variable with an empty domain in $\text{AC}(P)$, denoted $\text{AC}(P) = \perp$, then P is clearly unsatisfiable. $P|_S$ with $S \subset \{X = a \mid X \in \mathcal{X} \wedge a \in \text{dom}(X)\}$ is the constraint network¹ obtained from P by restricting the domain of X to the singleton $\{a\}$ for any variable assignment $X = a \in S$.

Definition 2 Let $P = (\mathcal{X}, \mathcal{C})$ be a constraint network, $X \in \mathcal{X}$ and $a \in \text{dom}(X)$. (X, a) is said to be singleton arc consistent iff $\text{AC}(P|_{X=a}) \neq \perp$. P is said to be singleton arc consistent iff $\forall X \in \mathcal{X}$, $\text{dom}(X) \neq \emptyset$ and $\forall a \in \text{dom}(X)$, (X, a) is singleton arc consistent.

$\mathcal{X}$ will be called the domain of $P = (\mathcal{X}, \mathcal{C})$. We will note $(X, a) \in P$ (respectively, $(X, a) \notin P$) iff $X \in \mathcal{X}$ and $a \in \text{dom}(X)$ (respectively, $a \notin \text{dom}(X)$).

3 Overview of SAC algorithms

The first algorithm that has been proposed to establish singleton arc consistency is called SAC-1 [Debruyne and Bessière, 1997b]. The principle of this algorithm is to check the singleton arc consistency of all variables whenever a value is detected singleton arc inconsistent and then removed. Worst-case space and time complexities of SAC-1 are respectively

¹It will be assumed that $X = a \in S \wedge Y = b \in S \Rightarrow X \neq Y$ and, for convenience, $P|_{\{X=a\}}$ will be simply denoted by $P|_{X=a}$.

$O(md)$ and $O(mn^2d^4)$ where n denotes the number of variables, d the size of the largest domain and m the number of constraints.

A second algorithm, denoted SAC-2, has been proposed by [Bartak, 2004]. The idea is to check (again) the singleton arc consistency of a value (Y, b) after the removal of a value (X, a) only if (X, a) does not support (Y, b) , i.e., does not belong to $\text{AC}(P|_{Y=b})$. Hence, this algorithm allows avoiding usefulness singleton checks by recording, for each value, the set of values supported by it. As expected and supported by the experimentation of [Bartak, 2004], SAC-2 offers a significant improvement of practical time efficiency with respect to SAC-1. Worst-case space and time complexities of SAC-2 are respectively $O(n^2d^2)$ and $O(mn^2d^4)$.

[Bessière and Debruyne, 2004] have remarked that SAC-2 does not present any improvement in terms of worst-case time complexity because whenever the singleton arc consistency of a value (X, a) must be checked again, one has to perform the arc consistency enforcement on $P|_{X=a}$ from scratch. In other words, SAC-2 does not exploit the incrementality of arc consistency. An arc consistency algorithm is said incremental if its worst-case time complexity is the same when it is applied one time on a given network P and when it is applied up to nd times on P where, between two consecutive executions, at least one value has been deleted. All current arc consistency algorithms are incremental. To benefit from the incrementality of arc consistency, [Bessière and Debruyne, 2004; 2005] have proposed a new algorithm, SAC-OPT, that duplicates the original constraint network into nd dedicated constraint networks, one for each value (X, a) of the instance. Simply, whenever the singleton consistency of a value (X, a) must be checked, the dedicated constraint network is used. Worst-case space and time complexities of SAC-OPT are respectively $O(mnd^2)$ and $O(mnd^3)$ which is the best time complexity that can be expected from an algorithm enforcing singleton arc consistency [Bessière and Debruyne, 2004].

Finally, from the observation that a space complexity in $O(mnd^2)$ prevents the use of SAC-OPT on large constraint networks, [Bessière and Debruyne, 2005] have proposed another algorithm called SAC-SDS which represents a trade-off between time and space. With respect to each value, only the domain (called SAC-support) is recorded as well as a propagation list used for arc consistency. In return, the data structures required to establish arc consistency are no more dedicated but shared. An experimental study on random instances have highlighted the good performance of this algorithm. Worst-case space and time complexities of SAC-SDS are respectively $O(n^2d^2)$ and $O(mnd^4)$.

4 SAC-3

All algorithms previously mentioned involve performing a breadth-first search up to a depth equal to 1. Each branch (of size 1) of this search corresponds to check the singleton arc consistency of a value, and allows removing this value if an inconsistency is found (after establishing arc consistency). One alternative is to check the singleton arc consistency of a value in the continuity of previous checks. In other words, we can try to build less branches of greater sizes using a greedy

search (where at each step, arc consistency is maintained). As long as, for a current branch, no inconsistency is found, we try to extend it. When an inconsistency is found, either the branch is of size 0 and a value is detected inconsistent, or all but last variable assignments correspond to singleton arc consistent values. This last statement relies on Proposition 1.

Proposition 1 *Let $P = (\mathcal{X}, \mathcal{C})$ be a constraint network and let $S \subset \{X = a \mid X \in \mathcal{X} \wedge a \in \text{dom}(X)\}$. If $\text{AC}(P|_S) \neq \perp$ then any pair (X, a) such that $X \in \mathcal{X}$ and $\text{dom}(X) = \{a\}$ in $\text{AC}(P|_S)$ is singleton arc consistent.*

Proof. If $\text{AC}(P|_S) \neq \perp$ then, clearly any element $X = a \in S$ is singleton arc consistent. It is a consequence of the monotony of arc consistency. One should observe that we can also find some values (Y, b) such that $Y \in \mathcal{X}$ and $\text{dom}(Y) = \{b\}$ in $\text{AC}(P|_S)$ with $Y = b \notin S$. These values are also clearly singleton arc consistent. $\square$

As mentioned in the proof above, some values can be detected singleton arc consistent while checking the singleton arc consistency of another one(s). Proposition 1 can then be seen as a generalization of Property 2 in [Chmeiss and Sais, 2000], and is also related to the exploitation of singleton-valued variables in [Sabin and Freuder, 1997].

Although the primary goal of our approach was to exploit incrementality of arc consistency, other nice features have been observed. Indeed, using a greedy search, one may find solutions and one can learn from conflicts by recording no-goods or weighting failure culprits.

Below, we give the description of a first algorithm that uses a greedy search in order to establish singleton arc consistency. The description is given in the context of using an underlying coarse-grained arc consistency algorithm (e.g. AC3 [Mackworth, 1977] or AC3.2/3.3 [Lecoutre *et al.*, 2003]) with a variable-oriented propagation scheme.

First, let us introduce some notations. If $P = (\mathcal{X}, \mathcal{C})$, then $\text{AC}(P, Q)$ with $Q \subseteq \mathcal{X}$ means enforcing arc consistency on P from the given propagation set Q . For a description of AC, see, for instance, the function *propagateAC* in [Bessière and Debruyne, 2005]. Q_{sac} is the set of values whose singleton arc consistency must be checked. A branch corresponds to a set of values that have been assigned. For any set of values $S \subseteq \{(X, a) \mid X \in \mathcal{X} \wedge a \in \text{dom}(X)\}$, $\text{vars}(S) = \{X \mid (X, a) \in S\}$. Finally, an instruction of the form $P_{\text{before}} \leftarrow P$ should not be systematically considered as a duplication of the problem. Most of the time, it correspond to store or restore the domain of a network (and the structures of the underlying arc consistency algorithm).

Algorithm 2 starts by enforcing arc consistency on the given network. Then, all values are put in the structure Q_{sac} and in order to check their singleton arc consistency, successive branches are built. The process continues until a fix-point is reached. Algorithm 1 allows building a branch by performing successive variable assignments while maintaining arc consistency (line 6). When an inconsistency is detected for a non empty branch, one has to put back the last value in Q_{sac} (line 12) since we have no information about the singleton arc consistency or inconsistency of this value. If the branch is empty, we have to manage the removal of a value and to reestablish arc consistency (lines 16 to 19). Note that

Algorithm 1 buildBranch()

```

1:  $br \leftarrow \emptyset$ 
2:  $P_{\text{before}} \leftarrow P$ 
3: consistent  $\leftarrow$  true
4: repeat
5:   pick and delete  $(X, a) \in Q_{\text{sac}}$  s.t.  $X \notin \text{vars}(br)$ 
6:    $P \leftarrow \text{AC}(P|_{X=a}, \{X\})$ 
7:   if  $P \neq \perp$  then
8:     add  $(X, a)$  to  $br$ 
9:   else
10:    consistent  $\leftarrow$  false
11:    if  $br \neq \emptyset$  then
12:      add  $(X, a)$  to  $Q_{\text{sac}}$ 
13:    end if
14: until not consistent  $\vee \text{vars}(Q_{\text{sac}}) - \text{vars}(br) = \emptyset$ 
15:  $P \leftarrow P_{\text{before}}$ 
16: if  $br = \emptyset$  then
17:   remove  $a$  from  $\text{dom}(X)$ 
18:    $P \leftarrow \text{AC}(P, \{X\})$ 
19:    $Q_{\text{sac}} \leftarrow Q_{\text{sac}} - \{(Y, b) \mid (Y, b) \in \text{dom}(P_{\text{before}}) - \text{dom}(P)\}$ 
20: end if

```

Algorithm 2 SAC-3($P = (\mathcal{X}, \mathcal{C})$: CSP)

```

1:  $P \leftarrow \text{AC}(P, \mathcal{X})$ 
2: repeat
3:    $P_{\text{before}} \leftarrow P$ 
4:    $Q_{\text{sac}} \leftarrow \{(X, a) \mid X \in \mathcal{X} \wedge a \in \text{dom}(X)\}$ 
5:   while  $Q_{\text{sac}} \neq \emptyset$  do
6:     buildBranch()
7:   until  $P = P_{\text{before}}$ 

```

no inconsistency is detected when a solution is found or when there is no way of extending the current branch. Finally, in order to maximally benefit from the incrementality of arc consistency, we have to build branches as long as possible. Hence (although not indicated in the algorithm), it is important to select first values $(X, a) \in Q_{\text{sac}}$ such that $a \in \text{dom}(X)$.

Proposition 2 *SAC-3 is a correct algorithm with a worst-case space complexity in $O(md)$ and a time complexity in $O(bmd^2)$ where b denotes the number of branches built by SAC-3.*

Proof. Correctness results from Proposition 1. If SAC-3 uses an optimal coarse-grained arc consistency algorithm such as AC3.2, then the overall space complexity is $O(md)$ since space complexity of AC3.2 is $O(md)$, the data structure Q_{sac} is $O(nd)$ and each branch built is $O(n)$. The overall time complexity is $O(bmd^2)$ since, due to incrementality, each branch built by the algorithm is $O(md^2)$. $\square$

Remark that b must include the “empty” branches that correspond to the detection of inconsistent SAC values. With respect to already singleton arc consistent constraint networks, Corollary 1 indicates that SAC-3 can outperform SAC-OPT and SAC-SDS (admitting then a time complexity in $O(mnd^3)$). More interestingly, it suggests that SAC-3 can outperform SAC-OPT and SAC-SDS on structured (not necessarily singleton arc consistent) instances that contain large under-constrained parts as can be expected in real-world applications.

Corollary 1 *SAC-3 admits a worst-case time complexity in $O(mn^2d^4)$ but, when applied to an already singleton arc consistent constraint network, SAC-3 admits a best-case² time complexity in $O(md^3)$ and a worst-case time complexity in $O(mnd^3)$.*

Proof. In the worst-case, $b = \frac{n^2d^2+nd}{2}$, hence, we obtain $O(mn^2d^4)$. When applied to an already singleton arc consistent constraint network, the best and worst cases correspond to branches of maximum size and of size 1 (1 consistent assignment followed by an inconsistent one), respectively. We have then respectively $b = d$ (all branches delivering a solution) and $b = nd$ branches. $\square$

5 SAC-3+

It is possible to improve the behaviour of the algorithm SAC-3 by recording the domain of the constraint networks obtained after each greedy run, that is to say, for each branch. When a value is removed, it is then possible to determine which previously built branches must be reconsidered. Indeed, if a removed value does not support a branch, i.e. does not belong to the domain associated with the branch, all values of the branch remain singleton arc consistent. On the other hand, if it supports a branch, we have to verify that the branch still remains valid by re-establishing arc consistency from the recorded domain. When a branch is no more valid, we have to delete it. In summary, SAC-3+ exploits incrementality as SAC-SDS does.

In order to manage domain and propagation of constraint networks corresponding to branches, we consider two arrays denoted $P[]$ and $Q[]$. For a given branch br , $P[br]$ corresponds to the constraint network associated with the branch br (in fact, we only need to record the domain of the constraint network) whereas $Q[br]$ contains the variables that have lost some value(s) and that should be considered when re-establishing arc consistency.

After enforcing arc consistency on the given network, Algorithm 6 builds successive branches by calling the function *buildBranch+*. Once the singleton arc consistency of all values of Q_{sac} have been tested, we have to check the validity of the branches that have been built and recorded in brs by a call to the function *checkBranches* since some values may have been deleted after a branch has been built. For each branch br , we re-establish arc consistency on $P[br]$ (line 2 of Algorithm 5) and in case of a domain wipe-out, we delete this branch and update Q_{sac} (lines 4 and 5).

Algorithm 4 differs from Algorithm 1 on two aspects. First, we need to record the domain of the constraint network corresponding to the branch that is built (line 22) and add this branch to brs (line 23). Note that if the last variable assignment entails a domain wipe-out, P_{store} is not updated (line 9). For the implementation, $P[br]$ can be directly set (backtracking one step if necessary) without any duplication of domain. Second, after re-establishing arc consistency (line 19), all values that have been removed including the singleton arc inconsistent one (line 18) must be taken into account in order

²Even if the worst-case time complexity of the underlying arc consistency algorithm is considered.

Algorithm 3 update(set : Set of Values)

```

1:  $Q_{sac} \leftarrow Q_{sac} - set$ 
2: for each  $br \in brs$  do
3:   for each  $(X,a) \in set$  do
4:     if  $(X,a) \in P[br]$  then
5:       remove  $(X,a)$  from  $P[br]$ 
6:       add  $X$  to  $Q[br]$ 
7:   endif
```

Algorithm 4 buildBranch+()

```

1:  $br \leftarrow \emptyset$ 
2:  $P_{before} \leftarrow P$ 
3: consistent  $\leftarrow$  true
4: repeat
5:   pick and delete  $(X,a) \in Q_{sac}$  s.t.  $X \notin vars(br)$ 
6:    $P \leftarrow AC(P|_{X=a}, \{X\})$ 
7:   if  $P \neq \perp$  then
8:     add  $(X,a)$  to  $br$ 
9:      $P_{store} \leftarrow P$ 
10:  else
11:    consistent  $\leftarrow$  false
12:    if  $br \neq \emptyset$  then
13:      add  $(X,a)$  to  $Q_{sac}$ 
14:    end if
15: until not consistent  $\vee vars(Q_{sac}) - vars(br) = \emptyset$ 
16:  $P \leftarrow P_{before}$ 
17: if  $br = \emptyset$  then
18:   remove  $a$  from  $dom(X)$ 
19:    $P \leftarrow AC(P, \{X\})$ 
20:   update( $\{(Y,b) | (Y,b) \in dom(P_{before}) - dom(P)\}$ )
21: else
22:    $P[br] \leftarrow P_{store}$ 
23:   add  $br$  to  $brs$ 
24: end if
```

Algorithm 5 checkBranches()

```

1: for each branch  $br \in brs$  do
2:    $P[br] \leftarrow AC(P[br], Q[br])$ 
3:   if  $P[br] = \perp$  then
4:      $Q_{sac} \leftarrow Q_{sac} \cup br$ 
5:     remove  $br$  from  $brs$ 
6:   end if
7: end for
```

Algorithm 6 SAC-3+($P = (\mathcal{X}, \mathcal{C})$: CSP)

```

1:  $P \leftarrow AC(P, \mathcal{X})$ 
2:  $brs \leftarrow \emptyset$ 
3:  $Q_{sac} \leftarrow \{(X,a) | X \in \mathcal{X} \wedge a \in dom(X)\}$ 
4: while  $Q_{sac} \neq \emptyset$  do
5:   while  $Q_{sac} \neq \emptyset$  do
6:     buildBranch+()
7:     checkBranches()
8:   end while
```

to update the state of all branches (line 20). For each branch (line 2 of Algorithm 3), we have to remove these values (line 5) and update the propagation list (line 6).

Proposition 3 *SAC-3+ is a correct algorithm with a space complexity in $O(b_{max}nd + md)$ and a time complexity in $O(bmd^2)$ where b_{max} denotes the maximum number of branches recorded by SAC-3+ and b denotes the number of times a branch is built or checked by SAC-3+.*

Proof. Correctness comes from Proposition 1 and the fact that once, the singleton arc consistency of all values have been checked and some branches recorded, one verify that the property still holds by calling *checkBranches*. In addition to the space requirement in $O(md)$ of the underlying optimal coarse-grained arc consistency algorithm, it is necessary to record the domain of the constraint networks corresponding to the valid branches that have been built. As recording a domain is in $O(nd)$, we obtain $O(b_{max}nd + md)$. $\square$

Remark that Corollary 1 also holds for SAC-3+. However, one should be optimistic about the average time complexity of this algorithm since it avoids building new branches when unnecessary.

6 Experiments

To prove the practical interest of the algorithms introduced in this paper, we have implemented them as well as the algorithms SAC-1 and SAC-SDS, the latter being considered as the most current efficient SAC algorithm [Bessière and Debruyne, 2005]. We have used AC3.2 [Lecoutre *et al.*, 2003] as an underlying arc consistency algorithms. We have conducted an experimentation on a PC Pentium IV 2,4GHz 512Mo under Linux with respect to different classes of random, academic and real-world instances. Performances have been measured in terms of the number of singleton arc consistency checks (#scks) and the cpu time in seconds (cpu). For information, is also given, for each instance, the number (# $\times$) of values removed by any SAC algorithm (when # $\times$ =0, it means that the instance is initially singleton arc consistent).

First, we have experimented the two classes of random binary CSP instances introduced in [Bessière and Debruyne, 2005]. However, due to lack of space, we only present the figure depicting results about the class $(100,20,0.05,t)$ corresponding to sparse constraint networks with 100 variables, 20 values per domain and a density of 0.05 (i.e., 248 constraints). t denotes the constraint tightness, i.e., the proportion of unallowed tuples in the relations associated with the constraints.

In Figure 1, we can observe that when $t < 0.6$ (the beginning of a phase transition), SAC-3 and SAC-3+ have the same behaviour and outperform SAC-1 and SAC-SDS. In the phase transition, SAC-3 and SAC-SDS respectively become the worst and the best approaches. For complete networks of the class $(100,20,1,t)$, similar results (not depicted here) are obtained, and at the pic of difficulty, SAC-SDS (181 s) is about three times more efficient than SAC-1 (478 s) and SAC-3 (553 s) and two times more efficient than SAC-3+ (307 s). It is not really a surprise since the generated instances have no structure, which corresponds to the worst-case for SAC-3 and SAC-3+ as the average size of the branches that are built (at the critical point) is quite small (≈ 3).

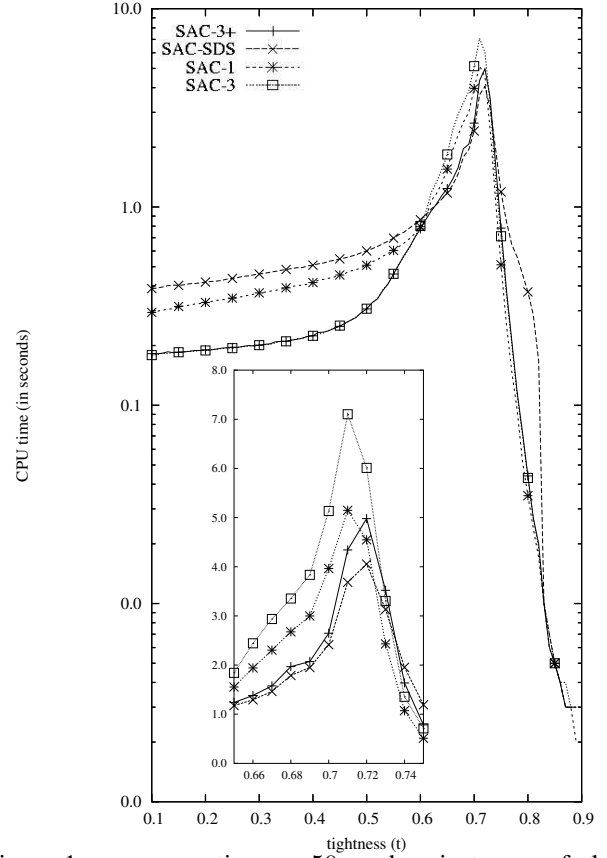


Figure 1: mean cpu time on 50 random instances of class $(100,20,0.05,t)$ at each value of t

Next, we have dealt with the following academic instances:

- two chessboard coloration instances, denoted *cc-20-2* and *cc-20-3*, involving quaternary constraints,
- two Golomb ruler instances, denoted *gr-34-8* and *gr-34-9*, involving binary and ternary instances,
- two prime queen attacking instances, denoted *qa-5* and *qa-6*, involving only binary constraints.

Table 1 shows that SAC-3, and especially SAC-3+, have on some instances a much better behaviour than SAC-1 and SAC-SDS. Roughly speaking, it can be explained by the fact that such instances have some regular structure.

Next, we have tested real-world instances, taken from the FullIRLFAP archive, which contains instances of radio link frequency assignment problems. Table 2 shows the results obtained on some representative instances. As expected, on already singleton arc consistent instances (*scen02*, *graph14*), a significant improvement is obtained. But it is also true for the other instances as they contain large under-constrained parts. It clearly appears that, on such structured instances, using SAC-3 and SAC-3+ is the best approach, especially as SAC-SDS has been out of memory on some instances.

It is interesting to note that SAC-3 and SAC-3+ can be much faster than SAC-1 and SAC-SDS even if the number of singleton checks (see for example, *cc-20-2* and *scen02*) is similar. It results from the exploitation of the incrementality of arc consistency (when building branches). Another point

		SAC-1	SAC-SDS	SAC-3	SAC-3+
cc-20-2 (#x=0)	cpu	14.44	14.43	3.46	3.48
	#scks	800	800	819	819
cc-20-3 (#x=0)	cpu	22.61	22.71	7.01	7.02
	#scks	1200	1200	1200	1200
gr-34-8 (#x=351)	cpu	66.08	17.43	38.28	18.62
	#scks	6335	3340	5299	1558
gr-34-9 (#x=513)	cpu	111.44	31.29	91.50	32.03
	#scks	8474	4720	11017	2013
qa-5 (#x=9)	cpu	2.47	2.50	.93	.96
	#scks	622	622	732	732
qa-6 (#x=48)	cpu	27.55	14.34	8.23	4.38
	#scks	2523	1702	2855	1448

Table 1: Academic instances

		SAC-1	SAC-SDS	SAC-3	SAC-3+
scen02 (#x=0)	cpu	20.97	20.73	4.09 (16)	4.08 (16)
	#scks	8004	8004	8005	8005
scen05 (#x=13814)	cpu	11.79	20.03	1.55 (1)	1.87
	#scks	6513	4865	4241	2389
graph03 (#x=1274)	cpu	215.88	136.93	74.97	39.10
	#scks	20075	17069	22279	8406
graph10 (#x=2572)	cpu	1389.59	-	675.64	349.53
	#scks	74321	-	82503	29398
graph14 (#x=0)	cpu	154.16	-	31.17 (10)	32.72 (10)
	#scks	36716	-	36719	36719

Table 2: RLFAP instances

to be mentioned is that some solutions (enclosed in brackets near cpu time) have been found during the inference process on some instances. For example, 16 solutions have been found on *scen02* by SAC-3 and SAC-3+.

Finally, we have experimented some realistic scheduling problems. We have selected 2 representative instances from the set of 60 job shop instances proposed by [Sadeh and Fox, 1996]. It is quite interesting to note (see Table 3) that SAC-3 and SAC-3+ have found solution(s) to these two instances quite more efficiently than MAC (when run to find one solution). It can be explained by the restart aspect of both algorithms.

		SAC-1	SAC-SDS	SAC-3	SAC-3+	MAC
js-1 (#x=0)	cpu	29.61	29.58	18.17 (4)	19.04 (4)	181.68
	#scks	5760	5760	6029	6029	-
js-2 (#x=0)	cpu	40.40	38.58	31.13 (1)	31.07 (1)	211.52
	#scks	6315	6315	6718	6718	-

Table 3: Job-shop instances

7 Conclusion

Establishing singleton arc consistency can be justified if it is effective (allows some filtering). On the contrary, applying a SAC algorithm on a constraint network that is already singleton arc consistent is a problem as it may involve a large waste of time. The two algorithms, SAC-3 and SAC-3+, introduced in this paper combine inference and search and can be understood as an answer to this problem. Indeed, when an instance is under-constrained or, more generally, contains easy large parts, as can be expected in real-world applications, exploiting search during inference can pay off. It has been confirmed

by our experimentation. Besides, one should have noted the limited space requirement of both algorithms which makes them applicable on large constraint networks.

We believe that this approach deserves further investigation in order to determine if it could be applied to other local consistencies and, to what extent, maintaining SAC during search could be a viable alternative to MAC.

Acknowledgments

This paper has been supported by the CNRS, the “programme COCOA de la Région Nord/Pas-de-Calais” and by the “IUT de Lens”.

References

- [Bartak, 2004] R. Bartak. A new algorithm for singleton arc consistency. In *Proceedings of FLAIRS'04*, 2004.
- [Bessière and Debruyne, 2004] C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency. In *Proceedings of ECAI'04 workshop on modeling and solving problems with constraints*, pages 20–29, 2004.
- [Bessière and Debruyne, 2005] C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, 2005.
- [Chmeiss and Sais, 2000] A. Chmeiss and L. Sais. About the use of local consistency in solving CSPs. In *Proceedings of ICTAI'00*, pages 104–107, 2000.
- [Debruyne and Bessière, 1997a] R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proc. of CP'97*, pages 312–326, 1997.
- [Debruyne and Bessière, 1997b] R. Debruyne and C. Bessière. Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.
- [Dechter, 2003] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
- [Lecoutre et al., 2003] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.
- [Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:118–126, 1977.
- [Prosser et al., 2000] P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proceedings of CP'00*, pages 353–368, 2000.
- [Sabin and Freuder, 1994] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the PPCPA'94*, 1994.
- [Sabin and Freuder, 1997] D. Sabin and E. Freuder. Understanding and improving the MAC algorithm. In *Proceedings of CP'97*, 1997.
- [Sadeh and Fox, 1996] N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86:1–41, 1996.

Identifying Conflicts in Overconstrained Temporal Problems

Mark H. Liffiton, Michael D. Moffitt, Martha E. Pollack, and Karem A. Sakallah

Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor 48109-2122
{liffiton, mmoffitt, pollackm, karem}@eecs.umich.edu

Abstract

We describe a strong connection between maximally satisfiable and minimally unsatisfiable subsets of constraint systems. Using this relationship, we develop a two-phase algorithm, employing powerful constraint satisfaction techniques, for the identification of conflicting sets of constraints in infeasible constraint systems. We apply this technique to overconstrained instances of the Disjunctive Temporal Problem (DTP), an expressive form of temporal constraint satisfaction problems. Using randomly-generated benchmarks, we provide experimental results that demonstrate how the algorithm scales with problem size and constraint density.

1 Introduction

Many AI problems can be cast in terms of Constraint-Satisfaction Problems (CSPs), and a large number of systems have been developed to efficiently compute solutions to sets of constraints. However, not every set of constraints is satisfiable. This paper reports on a technique for efficiently identifying sets of conflicting constraints in an overconstrained problem. We focus on temporal constraint problems, which play a prominent role in planning, scheduling, and other applications of interest in AI. Our approach is particularly well-suited to temporal problems, in which conflicts among constraints can be resolved by weakening, rather than completely abandoning, constraints.

We build on our previous work [Liffiton and Sakallah, 2005] on identifying Minimally Unsatisfiable Subsets of constraints (MUSes). Given a set of constraints C , an MUS of C is a subset of C that is (1) unsatisfiable and (2) minimal, in the sense that removing any one of its elements makes the rest of the MUS satisfiable. Each MUS thus provides information about a conflict that must be addressed to solve the given CSP. In general, an arbitrary CSP may contain multiple MUSes, and all of them must be resolved by constraint relaxation before the CSP can be solved. Identifying the MUSes of a CSP makes it possible to reason about how to weaken conflicting constraints to make a solution feasible.

Prior work on MUSes has largely been restricted to algorithms for finding a single MUS. In addition, these algo-

rithms have been specific to particular types of constraint systems, e.g., Boolean satisfiability ([Bruni and Sassano, 2001; Zhang and Malik, 2003; Oh *et al.*, 2004] and others) or linear programming [Chinneck, 1996], and the techniques developed often cannot be readily generalized to other types of constraints. In contrast, we explore a deep connection between maximal satisfiability and minimal unsatisfiability to develop techniques that can be used to find *all* MUSes of a constraint system, regardless of constraint type. In this paper, we demonstrate the approach by applying it to Disjunctive Temporal Problems (DTPs) [Stergiou and Koubarakis, 1998], a particularly expressive form of temporal constraints. Our previous work focused on finding MUSes of Boolean satisfiability instances, which have applications in circuit design and verification. DTPs provide a richer space for using MUSes, specifically because their constraints can be relaxed without any change in structure, something that is not possible with Boolean variables and constraints.

In Section 2 of the paper, we define DTPs and discuss the motivation for finding their MUSes further. Section 3 describes the relationship between maximal satisfiability and minimal unsatisfiability. Section 4 contains the specific algorithms we employed to exploit this relationship for finding MUSes of DTPs. Empirical results are given in Section 5, and Section 6 closes with conclusions and directions for future work.

2 Disjunctive Temporal Problems

A *Disjunctive Temporal Problem* (DTP) is a constraint satisfaction problem defined by a pair $\langle X, C \rangle$, where X is a set of real variables, designating time points, and C is a set of constraints of the form: $c_{i1} \vee c_{i2} \vee \dots \vee c_{in}$, where each c_{ij} is of the form $x - y \leq b$; $x, y \in X$ and $b \in \mathbb{R}$. DTPs are thus a generalization of Simple Temporal Problems (STPs), in which each constraint is limited to a single inequality [Dechter *et al.*, 1991]. A solution to a DTP is an assignment of values to time points such that all constraints are satisfied.

Several algorithms have been developed for solving DTPs [Stergiou and Koubarakis, 1998; Armando *et al.*, 1999; Oddi and Cesta, 2000; Tsamardinos and Pollack, 2003]. Typically, these algorithms transform the problem into a *meta-CSP*, in which the original DTP is viewed as a collection of alternative STPs. Using this approach, the algorithm selects a single disjunct from each constraint of a given DTP. The re-

sulting set forms an STP, called a *component STP*, which can then be checked for consistency in polynomial time using a shortest-path algorithm. Clearly, a DTP D is consistent if and only if it contains at least one consistent component STP. Furthermore, any solution to a consistent component STP of DTP D is also a solution to D itself. Consequently, it is standard in the DTP literature to consider any consistent component STP to be a solution of the DTP to which it belongs.

A number of pruning techniques can be used to focus the search for a consistent component STP of a DTP. These include conflict-directed backjumping, removal of subsumed variables, and semantic branching. The DTP solver Epilitis [Tsamardinos and Pollack, 2003] integrated all these techniques, in addition to no-good recording. Epilitis (and other traditional DTP solvers) perform total constraint satisfaction—that is, their objective is to find a solution that satisfies all the constraints of a DTP. In the event that a DTP is inconsistent, these solvers are capable of detecting such infeasibility, but incapable of providing partial solutions that come close to satisfying the problem. In response, the DTP solver Maxilitis [Moffitt and Pollack, 2005] was designed to find solutions that maximize the number of satisfied constraints.

Sometimes the number of satisfied constraints is not the best measure of a solution; it may instead be important to reason about exactly which constraints are violated, as well as the extent to which they have been violated. We illustrate with the following very small DTP, which we use as an example throughout the paper:

$$\begin{aligned} C_1 &: \{c_{11} : (x - y \leq 1)\} \\ C_2 &: \{c_{21} : (y - x \leq -2)\} \\ C_3 &: \{c_{31} : (y - x \leq -3)\} \vee \{c_{32} : (z - y \leq 1)\} \\ C_4 &: \{c_{41} : (y - z \leq -2)\} \end{aligned}$$

Note that this DTP is unsatisfiable. The set of constraints $\{C_1, C_2\}$ is inconsistent, and the same is true for $\{C_1, C_3, C_4\}$. A standard DTP solver would simply report that the problem was unsatisfiable, while a system like Maxilitis would find a solution that maximizes the number of satisfied constraints—in this case, a solution that satisfies $\{C_2, C_3, C_4\}$ and not C_1 . But for many applications, it might be important instead to maintain some control over the relationship between x and y that is expressed in C_1 . If we knew the source of the conflict involving C_1 , we could reason about how to weaken C_1 and the constraints with which it conflicts. For example, we could decide to change the bound on C_1 to 2, or to change the bounds on both C_1 and C_2 to 1.5. We could reason about similar relaxations of C_1 , C_3 , and C_4 to make them jointly feasible. But in order to do this, we must have identified which constraints are in conflict with one another, and this is precisely what the MUSes tell us.

This ability to relax constraints without completely removing them makes DTPs a much more interesting domain for finding MUSes than, for example, systems of Boolean constraints. An infeasible Boolean satisfiability instance may be made feasible by completely removing constraints, and such changes are indicated directly by techniques that identify a maximal satisfiable subset of constraints. An infeasible DTP, however, can be made consistent by altering the bounds

of inequalities, without modifying the structure of the problem. MUSes provide information needed to determine these changes.

3 Maximal Satisfiability and Minimal Unsatisfiability

Our techniques for extracting MUSes are derived from a deep relationship between maximal satisfiability and minimal unsatisfiability. The Maximal Constraint Satisfaction problem (Max-CSP) is an optimization problem on a constraint system C that has the goal of finding an assignment to the variables of C that satisfies as many constraints as possible; this is the solution that Maxilitis finds (e.g., $\{C_2, C_3, C_4\}$ for our example problem).

While Max-CSP is defined in terms of the cardinality of a satisfiable subset of constraints, the definition can be relaxed to have *inaugmentability* as the goal instead. Thus, while we can define $\text{Max-CSP}(C)$ as $\{m \subseteq C : |m| \text{ is maximal, } m \text{ is satisfiable}\}$, we can define a new problem, *Maximally Satisfiable Subset* (MSS). The definition of the set of MSSes follows, with the set of MUSes defined similarly for comparison:

$$\begin{aligned} \text{MSSes}(C) &= \left\{ m \subseteq C : m \text{ is satisfiable, and } \forall c \in (C \setminus m), m \cup \{c\} \text{ is unsatisfiable} \right\} \\ \text{MUSes}(C) &= \left\{ m \subseteq C : m \text{ is unsatisfiable, and } \forall c \in m, m \setminus \{c\} \text{ is satisfiable} \right\} \end{aligned}$$

Each MSS is a subset of C that is satisfiable and *inaugmentable*; adding any other constraints in C to such a subset will render it unsatisfiable. Notice from the definition that $\text{MSSes}(C)$ and $\text{MUSes}(C)$ are essentially duals of one another! An MSS is satisfiable and cannot be made larger, and an MUS is *unsatisfiable* and cannot be made *smaller*. This relationship is more than cosmetic; we will show how one set is actually an implicit encoding of the other. Note also that any Max-CSP solution is also an MSS; maximal cardinality implies *inaugmentability*. However, MSSes may not all have maximal cardinality, as illustrated in the example DTP. One MSS of the example is $\{C_2, C_3, C_4\}$, corresponding to the Max-CSP solution. $\{C_1, C_4\}$ is another MSS (adding either C_2 or C_3 makes it unsatisfiable), though smaller.

Now consider C_1 , the constraint not included in the Max-CSP solution to our example. Removing C_1 from the DTP makes the remaining system satisfiable. In general, given any MSS, the set of constraints not included in that MSS provides an irreducible “fix” for the original infeasible system; removing these constraints makes it satisfiable. Therefore, we define a “CoMSS” as the complement of an MSS, and the set $\text{CoMSSes}(C)$ as:

$$\text{CoMSSes}(C) = \{m \subseteq C : (C \setminus m) \in \text{MSSes}(C)\}$$

This complementary view of $\text{MSSes}(C)$ reveals a more meaningful connection between maximal satisfiability and minimal unsatisfiability. Recall that the presence of any MUS in a constraint system C makes C infeasible. Therefore, to make C feasible, every MUS in C must be neutralized by relaxing or removing at least one constraint from it. Any

CoMSSes(C)	C_1	C_2	C_3	C_4
$\{C_1\}$	X			
$\{C_2, C_3\}$		X	X	
$\{C_2, C_4\}$		X		X

$$\begin{aligned}
\text{MUSes}(C) &= (C_1)(C_2 \vee C_3)(C_2 \vee C_4) \\
&= C_1 C_2 \vee C_1 C_3 C_4 \\
&= \{\{C_1, C_2\}, \{C_1, C_3, C_4\}\}
\end{aligned}$$

MUSes(C)	C_1	C_2	C_3	C_4
$\{C_1, C_2\}$	X	X		
$\{C_1, C_3, C_4\}$	X		X	X

$$\begin{aligned}
\text{CoMSSes}(C) &= (C_1 \vee C_2)(C_1 \vee C_3 \vee C_4) \\
&= C_1 \vee C_2 C_3 \vee C_2 C_4 \\
&= \{\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}\}
\end{aligned}$$

Figure 1: Covering Problems Linking CoMSSes(C) and MUSes(C)

CoMSS is one such set: a CoMSS m is an irreducible set of constraints whose removal makes C satisfiable. Thus, every CoMSS contains at least one constraint from every MUS of C . For temporal problems, achieving satisfiability may not require removing every constraint in a CoMSS, but it does require relaxing all of them. Of course, removing a constraint is equivalent to extreme relaxation, in which the bound of an inequality in the constraint is set to infinity.

Every CoMSS is inherently a solution to a covering problem on MUSes(C). Specifically, it is a solution to the HITTING-SET problem [Garey and Johnson, 1990] on MUSes(C). Given a set C , and a collection of subsets of C , a hitting set of the collection is a subset of C that contains at least one element from each subset in the collection. In this case, the set C is the set of constraints in a constraint system, the collection of subsets of C is the set MUSes(C), and the hitting set of MUSes(C) is a CoMSS of C .

Additionally, in line with the “duality” of MSSes and MUSes, every MUS is a hitting set of the set of CoMSSes. This is the key to extracting all MUSes of a constraint system: first compute the set of CoMSSes of C , then extract MUSes by finding hitting sets of the CoMSSes.

Figure 1 illustrates the relationship with covering problems that link CoMSSes(C) and MUSes(C) for our example DTP. The left table represents the covering problem of finding MUSes that cover the CoMSSes to generate MUSes(C). Each column is a constraint from the DTP, and each row is a single CoMSS. We can say that a constraint “covers” a CoMSS (marked with an ‘X’ in that row) if it is contained in the CoMSS. Each MUS is then an irreducible subset of the columns that covers all of the rows. In the example, these are found in a manner similar to Petrick’s Method from Boolean logic minimization: each row becomes a disjunction of the columns that cover that row, and the disjunctions are conjoined and simplified by the distributive rule. The right half of the figure shows how CoMSSes(C) can be generated in the exact same manner from MUSes(C).

If we are interested primarily in finding an arbitrary set of constraints to relax in order to make an overconstrained set C feasible, then we could simply compute CoMSSes(C): each element of CoMSSes(C) represents one set of constraints that, if sufficiently relaxed, will result in C being feasible. But if we instead want to reason about how to weaken particular constraints, we need to identify the individual sets of conflicting constraints, i.e., MUSes(C). In practice, it is eas-

Simplified-Musilitis-CoMSSes(A, U)

1. If ($U = \emptyset$)
2. $\text{newCoMSS} \leftarrow \{C \mid (C \leftarrow \epsilon) \in A\}$
3. $\text{CoMSSes} \leftarrow \text{CoMSSes} \cup \{\text{newCoMSS}\}$
4. $\text{backjump-to-}\epsilon \leftarrow \text{true}$
5. Return
6. $C \leftarrow \text{Select-Constraint}(U)$, $U' \leftarrow U - \{C\}$
7. For all values $c \in d(C)$
8. $A' \leftarrow A \cup \{C \leftarrow c\}$
9. If **consistent**(A')
10. **Simplified-Musilitis-CoMSS**(A', U')
11. If ($\text{backjump-to-}\epsilon = \text{true}$) Return
12. $A' \leftarrow A \cup \{C \leftarrow \epsilon\}$
13. Unless (**Assignment-Subsumed**($A', \text{CoMSSes}$))
14. **Simplified-Musilitis-CoMSS**(A', U')
15. $\text{backjump-to-}\epsilon \leftarrow \text{false}$

Figure 2: A simplified version of an algorithm for finding the CoMSSes of a DTP

ier to find maximally satisfiable subsets (and thus, CoMSSes) than to find MUSes directly. This follows from the relative simplicity of problems in NP (i.e., SAT) as compared to those in Co-NP (i.e., UNSAT). Thus, our approach will be to compute CoMSSes(C) and then find its irreducible hitting sets.

4 Algorithm Details

We now describe the details of our algorithm for finding MUSes. We use a serial decomposition of the task. First we find all of the CoMSSes of a DTP using an algorithm that borrows heavily from Maxilitis; this algorithm could be readily generalized for other types of constraints. Second, we use techniques for extracting MUSes that operate completely independently of how the CoMSSes were generated and the types of constraints involved. The two phases combine to form a solver, which we name Musilitis, that is capable of diagnosing the infeasibility of any given DTP. The solver is both sound, in that the sets of constraints it returns are all MUSes, and complete, in that it will find all MUSes of the given constraint system.

4.1 Finding CoMSSes(C)

In Figure 2, we present a simplified version of an algorithm for finding the set of all CoMSSes of a DTP. This backtrack-

ing search resembles in many ways the branch-and-bound algorithm employed by Maxilitis, in that it effectively searches through the space of all partial assignments of disjuncts to constraints. (Recall that we convert each disjunctive constraint in the original DTP into a meta-level variable, whose domain is the set of disjuncts.) The set A contains the current partial assignment, and the set U contains those constraints that have not yet been instantiated. Whenever a constraint is instantiated with an empty assignment (i.e., it is purposefully chosen to be left unsatisfied), such an assignment is labeled by the symbol ϵ .

To understand the algorithm, consider an example with 50 constraints. Suppose that we begin the search and find a solution in which two constraints, say C_6 and C_{28} , have been left unsatisfied. At this point, the search can prune any other solutions that would necessarily leave C_6 and C_{28} unsatisfied, as such candidates would be incapable of leading to an irredundant CoMSS. We handle this in our algorithm by means of a backjump flag that causes the search to immediately backjump to the deepest point at which the assignment to one of the unsatisfied constraints was made.

Lines 1 through 5 handle the case when a particular solution with all constraints instantiated has been found. The set of unsatisfied constraints (those assigned ϵ) is a CoMSS, and it is added to a global list of CoMSSes (initialized to $\emptyset$). At this point, the flag *backjump-to- ϵ* is toggled to avoid any further search within the current subtree whose solutions would be subsumed by the CoMSS just found.

If one or more constraints remain uninstantiated, line 6 selects a constraint to satisfy, and lines 7 – 11 attempt various disjuncts with which to satisfy it. However, if one of the recursive calls finds a solution, the *backjump-to- ϵ* flag may be enabled, causing the algorithm to bubble up the search tree until the deepest assignment of ϵ is reached.

Line 12 is where the algorithm attempts the empty assignment. If this assignment is not subsumed by any CoMSSes found in prior solutions (line 13), the algorithm will pursue this partial assignment (line 14), afterwards resetting the backjump flag in the event that a solution below was discovered (line 15). It should be noted that this version of the algorithm is not guaranteed to find only irredundant CoMSSes, but any redundancies can be easily removed in a postprocessing step.

A slight variation on this approach is to find CoMSSes in increasing order of size. By blocking CoMSSes found in increasing order, the algorithm ensures that all subsequent CoMSSes found are irredundant. Such an incremental algorithm can easily be achieved by setting an upper bound on the number of constraints violated, and repeatedly calling the given algorithm, each time increasing the upper bound by 1 until the problem is no longer satisfiable. While many searches are performed, the pruning allowed by the smaller CoMSSes may potentially save time otherwise spent finding redundant CoMSSes. This incremental algorithm proved to be extremely effective in Maxilitis, where the goal was simply that of maximal constraint satisfaction.

If implemented exactly as shown, the algorithm shown in Figure 2 would perform rather poorly. The pseudocode does not illustrate the use of forward checking, a vital tech-

Musilitis-ExtractMUS(CoMSSes)

1. $MUS \leftarrow \emptyset$
2. While (CoMSSes is not empty)
3. $curCoMSS \leftarrow \text{pop}(\text{CoMSSes})$
4. $newConstraint \leftarrow \text{pop}(curCoMSS)$
5. $MUS \leftarrow MUS \cup newConstraint$
6. For all $testConstraint \in curCoMSS$
7. For all $testCoMSS \in \text{CoMSSes}$
8. If $testCoMSS$ contains $testConstraint$
9. remove $testConstraint$ from $testCoMSS$
10. For all $testCoMSS \in \text{CoMSSes}$
11. If $testCoMSS$ contains $newConstraint$
12. remove $testCoMSS$ from CoMSSes
13. Return MUS

Figure 3: Extracting a single MUS from CoMSSes(C)

nique used in all DTP solving algorithms. It also does not demonstrate the use of two pruning techniques called *removal of subsumed variables* and *semantic branching* [Armando *et al.*, 1999] that are known to greatly improve the power of DTP solvers. Luckily, these methods remain available when finding CoMSSes of a DTP, and they are integrated into our Musilitis solver. Other common issues, such as variable and value ordering heuristics, also play a large role in guiding the search, although we do not discuss them here due to space limitations.

4.2 Obtaining MUSes(C)

Every MUS of C is an irreducible hitting set of CoMSSes(C). The decision version of the HITTING-SET problem (deciding if there exists a hitting set of cardinality less than or equal to k) is NP-Complete [Karp, 1972], but in this case we are interested in irreducibility, not cardinality. Combined with the property that no CoMSS is a subset of any other, this allows us to extract a single MUS via a greedy approach in polynomial time, without resorting to search. Figure 3 provides pseudocode for the algorithm.

The construction works by sequentially adding constraints to a forming MUS. Intuitively, an MUS is a set of constraints containing at least one constraint from every CoMSS such that every constraint is an essential element of the set. By “essential” we mean that removing a constraint from the MUS will leave at least one CoMSS unrepresented. Thus, every time a constraint is added, the remaining problem is modified to enforce the requirement that the constraint be essential. This is done by ensuring that the new constraint is the only representative of at least one CoMSS.

Lines 3 and 4 of **Musilitis-ExtractMUS** choose a CoMSS from the set and a single constraint, *newConstraint*, from that CoMSS. *newConstraint* is added to the MUS, and the remaining lines ensure that it is essential. Lines 6 through 9 remove the remaining constraints in the chosen CoMSS from all other CoMSSes in the input set. This prevents any of those constraints from being added in later iterations, which could make *newConstraint* non-essential. Then, lines 10-12 remove any CoMSSes containing *newConstraint*, because they are now represented in *MUS*. Following the modifications to

CoMSSes, the algorithm iterates. When CoMSSes is empty, the constructed set of constraints is a complete, exact MUS.

Extracting all of the MUSes from CoMSSes(C) requires searching for all irreducible hitting sets of CoMSSes(C). Due to the potential combinatorial explosion, the number of MUSes may be impractically large; however, in many cases the result is tractable.

One method for generating the complete set of MUSes from CoMSSes(C) uses the general form of the **Musilitis-ExtractMUS** algorithm in Figure 3. The particular MUS generated by that algorithm depends on the order in which CoMSSes and constraints are selected from CoMSSes(C); thus, by branching on those two decisions (lines 3 and 4), all possible MUSes can be generated. This can be accomplished with a recursive function that takes as input a) the remaining set of CoMSSes and b) the MUS currently being constructed in each branch of the recursion (these are initialized to CoMSSes(C) and the empty set, respectively). The branching is not ideal, and duplicate branches are both possible and quite common in practice, so pruning heuristics can be employed to reduce the search space.

5 Experimental Results

We benchmarked both phases of the MUS generation process (finding CoMSSes(C) and extracting MUSes(C) from it) using DTPs created by a random DTP generator used in testing previous DTP solvers [Stergiou and Koubarakis, 1998]. We collected performance data and analyzed the sets of CoMSSes and MUSes for DTPs over a range of DTP generator parameters.

The test case generator takes as arguments the parameters $\langle k, N, m, L \rangle$, where k is the number of disjuncts per constraint, N is the number of time points, m is the number of constraints, and L is the constraint width, i.e., a positive integer such that for each disjunct $x - y \leq b$, $b \in [-L, L]$ with uniform probability. In our experiments, we set $k = 2$, $N \in \{5, 6, 7\}$, and $L = 100$. A derived parameter R (the ratio of constraints over variables, m/N) was varied from 6 to 11. For each setting of N and R , 50 random problems were generated. (For example, we generated 50 problems for the case where N is 7 and R is 7; those problems have 7 temporal variables and 49 constraints each). These problems are likely to be highly overconstrained, with 30-77 constraints on only 5-7 time points. The size of these problems is dwarfed somewhat by those which Maxilitis can solve (e.g., $N = 20$), although one must keep in mind that the objective of maximal constraint satisfaction is much easier to achieve. The domains of the variables are integers instead of reals, which again is standard in DTP literature. Our implementation of Musilitis was developed in Java (finding CoMSSes) and C++ (extracting MUSes), and our experiments were conducted in Linux on a 2.2GHz Opteron processor with 8GB of RAM.

First, we compared the runtime of the two versions of the **Musilitis-CoMSS** algorithm described in Section 4.1—one which uses a single backtracking search to find all CoMSSes, and another which does so incrementally. The former requires an additional postprocessing step to remove all redundant CoMSSes, a procedure whose runtime was found to be

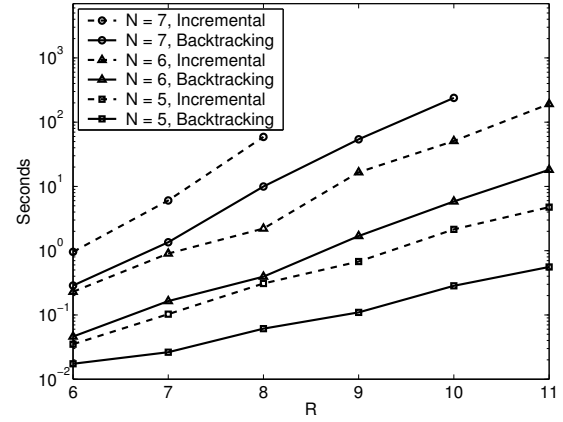


Figure 4: Median runtimes for finding CoMSSes(C)

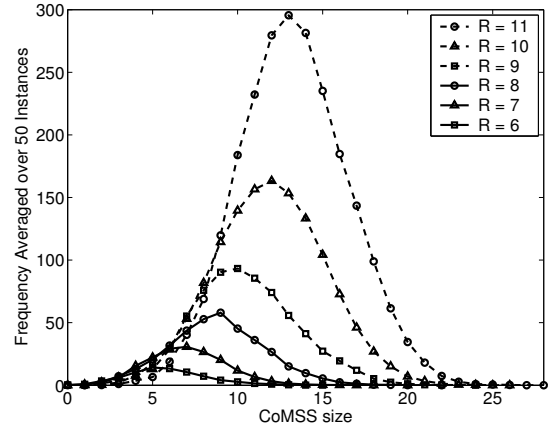


Figure 5: Distribution of CoMSS sizes ($N = 6$)

negligible. Figure 4 shows the median runtimes for both algorithms on all problem parameters. Not surprisingly, runtime increases both with size (N) and density (R). However, notice that the times for the single backtracking algorithm (labeled ‘Backtracking’) tend to be roughly an order of magnitude faster than those for the incremental method (labeled ‘Incremental’). This contrasts with the results reported in [Moffitt and Pollack, 2005], where the opposite effect was observed. The difference is that Maxilitis can stop at any iteration where a solution is found, as only a single solution of maximal cardinality is needed. Even though multiple searches are performed in the incremental mode, substantially more is pruned than in a single search that has no upper bound. In our case, iteration must continue until *all* CoMSSes of any size have been found. The combined runtime of these separate searches can easily exceed that of a single backtracking search.

Figure 5 shows the distribution of sizes of CoMSSes found by the **Musilitis-CoMSS** algorithm for the cases where $N = 6$ and R ranges from 6 to 11. Because our DTPs were randomly generated, these results are not particularly useful alone; however, they do provide valuable information when one considers the runtime of the MUS extraction phase of

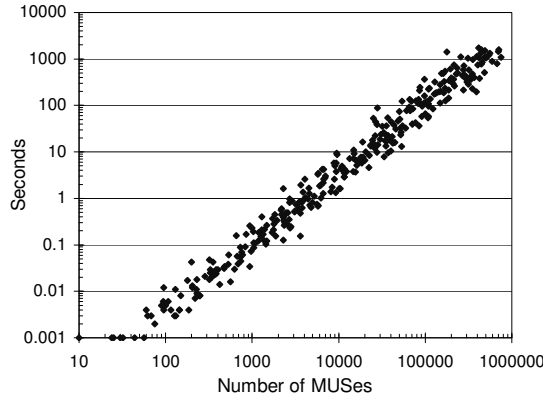


Figure 6: Runtimes of extracting MUSes(C) vs $|\text{MUSes}(C)|$

the Musilitis algorithm, since the set of CoMSSes is the input to this procedure. In the graph, we have taken the total frequency of each size of CoMSS found in all instances and divided this count by 50 (the number of problem instances). As one would expect, both the number and size of CoMSSes increases with the constraint density R .

The runtime of extracting all MUSes depends mainly on the number of MUSes present. In the extremely overconstrained benchmarks used here, this number is often very high, reaching into the hundreds of thousands. Figure 6 shows the runtime of the MUS extraction algorithm versus the number of MUSes found for instances with $N = 5$. Runtimes for any given number of MUSes are clustered within an order of magnitude of each other.

In some benchmarks, the MUS extraction reached a 30-minute timeout without finishing. In that time, however, it found as many as one million MUSes. The algorithm for extracting MUSes is inherently an anytime algorithm; MUSes are output as they are found, and if not all MUSes are required, it can be stopped at any time. Our experiments showed that the algorithm has reasonably good anytime properties: in one typical highly-constrained instance with $N = 6$ and $R = 9$, it generated more than 70,000 MUSes in the first minute, and the rate then gradually slowed. This property could be exploited in a system that interleaves MUS identification with constraint relaxation. Generally, if the constraint system is highly overconstrained, resolving one MUS is likely to resolve many others, since the same constraints are likely to appear in multiple MUSes.

6 Conclusions and Future Work

We have presented algorithms for deriving minimally unsatisfiable subsets of constraints in any infeasible constraint set, building on a strong relationship between maximal satisfiability and minimal unsatisfiability. The techniques are general, in that they are applicable to any type of constraint, and we have implemented them for Disjunctive Temporal Problems. The flexibility permitted by the constraints involved in these problems presents the opportunity for relaxation to be more fine-grained than simple constraint elimination, and MUSes provide the information needed to support this type of relaxation.

The connection between maximal satisfiability and minimal unsatisfiability we describe has been independently noted by [Bailey and Stuckey, 2005], who applied it to the problem of type-error diagnosis in software verification. Their implementation, however, differs from ours, in that they employ different algorithms in an interleaved approach as compared to our serial, two-phase algorithm. In [Liffiton and Sakallah, 2005], we performed an experimental comparison of their approach, adapted to Boolean satisfiability, with our own. We found that ours was consistently faster, and we identified several ways to combine ideas from both techniques for further improvements.

Further work also includes enhancing the efficiency of our algorithms. One possible direction would be to investigate tradeoffs between runtime and completeness or correctness. The requirements of finding all MUSes or of finding exact MUSes could be relaxed to decrease the runtime of the algorithms. Techniques exploiting domain knowledge or features of particular types of constraints could help improve the efficiency as well. More generally, it will be important to conduct studies that examine the performance of Musilitis on DTPs representing real-world problems, as opposed to the random problems studied in this paper. We currently have experiments underway to evaluate Musilitis on job-shop scheduling problems, and we will also be studying problems taken from the domains of schedule management and medical protocols.

The relationship between maximal satisfiability and minimal unsatisfiability is deserving of further attention. Our algorithms for extracting all MUSes are just one way of exploiting this relationship. Further understanding of infeasibility in constraint satisfaction problems, such as rich links between conflicts in basic constraints and in the high-level problem they represent, could result from additional research.

Finally, an important future goal is to employ the algorithms developed in this paper in a mixed-initiative constraint relaxation system that provides a user with the information needed to make principled decisions about how to weaken constraints so as to achieve feasibility.

Acknowledgments

This material is based upon work supported by the National Science Foundation under ITR Grant No. 0205288, the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010, and the Air Force Office of Scientific Research under Contract No. FA9550-04-1-0043. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the National Science Foundation (NSF), DARPA, the Department of Interior-National Business Center, or the United States Air Force.

References

- [Armando *et al.*, 1999] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European Conference on Planning*, pages 97–108, 1999.
- [Bailey and Stuckey, 2005] James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of

- constraints using hitting set dualization. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, volume 3350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
- [Bruni and Sassano, 2001] Renato Bruni and Antonio Sassano. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, Electronic Notes in Discrete Mathematics volume 9. Elsevier Science Pub., 2001.
- [Chinneck, 1996] John W. Chinneck. An effective polynomial-time heuristic for the minimum-cardinality IIS set-covering problem. *Annals of Mathematics and Artificial Intelligence*, 17:127–144, 1996.
- [Dechter *et al.*, 1991] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.
- [Garey and Johnson, 1990] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.
- [Karp, 1972] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [Liffiton and Sakallah, 2005] Mark H. Liffiton and Karem A. Sakallah. On finding all minimally unsatisfiable subformulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, Lecture Notes in Computer Science. Springer-Verlag, 2005.
- [Moffitt and Pollack, 2005] Michael D. Moffitt and Martha E. Pollack. Partial constraint satisfaction of disjunctive temporal problems. In *Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference (FLAIRS-2005)*, 2005.
- [Oddi and Cesta, 2000] Angelo Oddi and Amedeo Cesta. Incremental forward checking for the disjunctive temporal problem. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 108–112, 2000.
- [Oh *et al.*, 2004] Yoonna Oh, Maher N. Mneimneh, Zacher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41st annual conference on Design automation (DAC'04)*, pages 518–523. ACM Press, 2004.
- [Stergiou and Koubarakis, 1998] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 248–253, 1998.
- [Tsamardinos and Pollack, 2003] Ioannis Tsamardinos and Martha E. Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151(1-2):43–90, 2003.
- [Zhang and Malik, 2003] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. Presented at the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), 2003.

Why Minimax Works: An Alternative Explanation

Mitja Luštrek and Matjaž Gams

Jožef Stefan Institute
Department of Intelligent Systems
Jamova 39, 1000 Ljubljana, Slovenia
mitja.lustrek@ijs.si, matjaz.gams@ijs.si

Ivan Bratko

University of Ljubljana
Faculty of Computer and Information Science
Tržaška 25, 1000 Ljubljana, Slovenia
bratko@fri.uni-lj.si

Abstract

In game-playing programs relying on the minimax principle, deeper searches generally produce better evaluations. Theoretical analyses, however, suggest that in many cases minimaxing amplifies the noise introduced by the heuristic function used to evaluate the leaves of the game tree, leading to what is known as pathological behavior, where deeper searches produce worse evaluations. In most of the previous research, positions were evaluated as losses or wins. Dependence between the values of positions close to each other was identified as the property of realistic game trees that eliminates the pathology and explains why minimax is successful in practice. In this paper we present an alternative explanation that does not rely on value dependence. We show that if real numbers are used for position values, position values tend to be further apart at lower levels of the game tree, which leads to a larger proportion of more extreme positions, where error is less probable. Decreased probability of error in searches to greater depths is sufficient to eliminate the pathology and no additional properties of game trees are required.

1 Introduction

Most game-playing programs are based on the minimax principle. Such programs choose the best move by searching the game tree to a chosen depth, heuristically evaluating the leaves and then propagating their values to the root using the minimax principle. It is generally agreed that deeper searches produce better evaluations at the root. However, first attempts to explain this mathematically yielded the paradoxical result that minimaxing amplifies the error of the heuristic evaluations and that consequently deeper searches produce worse evaluations [Nau, 1979; Beal, 1980]. This phenomenon is called the minimax pathology [Nau, 1979].

It was evident that the setting of these mathematical analyses omitted some property of real games that eliminates the pathology. Several explanations were proposed, but eventually most researchers came to the conclusion that the property they were looking for is the similarity of positions close to each other [Bratko and

Gams, 1982; Beal, 1982; Nau, 1982; Scheucher and Kaindl, 1998; Luštrek, 2004].

In most research on the minimax pathology, true position values were losses or wins. This seems reasonable, since in games like chess, positions can indeed only be lost or won (or drawn). In practice, however, a game playing program needs an evaluation function that makes it possible to maintain a *direction* of play, gradually moving toward a win, not just maintaining a won position without achieving the final goal. This requires a multivalued or even real-valued evaluation function.

We introduce a minimax model using real-valued evaluation function and a way to interpret real values as losses and wins. This leads to a new, simpler explanation of the pathology. Namely, node values at lower levels of the game tree are more dispersed. If the error of the evaluation function, represented by normally distributed noise, is independent of the depth of search, it turns out that the error in terms of loss/win evaluations decreases with the depth of search, because larger dispersion means a larger proportion of more extreme positions where error is less likely. This is different from what the previous analyses assumed and is sufficient to eliminate the pathology.

The paper is organized as follows. Section 2 presents the minimax pathology and gives an overview of the attempts to explain it. Section 3 introduces a minimax model based on real-number position values. Section 4 shows why the model from section 3 is not pathological, while seemingly similar models used in previous research were. Section 5 explains whether minimax in general can be expected to behave non-pathologically. Section 6 concludes the paper and points out where further research is needed.

2 The Pathology and Related Work

The minimax pathology was discovered independently by Nau [1979] and Beal [1980]. Beal's basic minimax model made several assumptions:

1. game tree has a uniform branching factor;
2. nodes of the tree can have two values: loss and win;
3. node values are distributed so that at each level of the tree the proportion of losses for the side to move is the same;
4. node values within each level of the tree are independent of each other;

5. the error of heuristic evaluation at a node at the lowest level of search, being the probability of mistaking a loss for a win or vice versa, is independent of the depth of search and the true value of the node.

We proceed to present Beal's basic model, although our analysis is mostly based on later work. Negamax representation is used in this section, i.e. node values are viewed as lost or won from the perspective of the side to move. Let b be the branching factor of the tree, d the depth of search and k_i the probability of a node at i -th level being lost. Levels are numbered downwards: from 0 for root to d for the lowest level of search.

A node can only be lost if all of its descendants are won (for the opponent), so the relation between the values of k at consecutive levels is governed by equation (1).

$$k_i = (1 - k_{i+1})^b \quad (1)$$

Assumption 3 requires $k_i = k_{i+1}$, which results in $k_i = c_b$ for all i ; for example, $c_2 = 0.3820$.

Two types of evaluation error are possible: a loss can be mistaken for a win (false win) or a win for a loss (false loss). Let p_i and q_i be the probabilities of the respective types of error at i -th level. They are calculated according to equations (2) and (3).

$$p_i = \frac{1}{k_i} (1 - k_{i+1})^b (1 - (1 - q_{i+1})^b) = 1 - (1 - q_{i+1})^b \quad (2)$$

$$q_i = \frac{1}{1 - k_i} \sum_{j=1}^b \binom{b}{j} k_{i+1}^j (1 - k_{i+1})^{b-j} p_{i+1}^j (1 - q_{i+1})^{b-j} \quad (3)$$

It turns out that if the same $p_d = q_d$ are used in searches to all depths, the error in the root, defined as $p_0 k_0 + q_0 (1 - k_0)$, increases with d . "This result is disappointing," concluded Beal, since it was exactly the opposite of what he set out to show. His model must have been flawed in some way.

In the years following the discovery of the pathology, several researchers attempted to find the flaw in Beal's basic model by attacking its assumptions (1. through 5. at the beginning of this section).

1. Michon [1983] observed that the pathology depends on the probability distribution of branching factor: game trees with uniform branching factor tend to be pathological, while game trees with, for example, geometrically distributed branching factor do not. It is not known whether real games have any of the non-pathological distributions, though.
2. Bratko and Gams [1982] were the first to experiment with multivalued evaluations. They assigned no particular meaning to different values, which resulted in behavior similar to the one with two values, i.e. pathological. Pearl's results were similar [1983]. Scheucher and Kaindl [1998] and Luštrek [2004] also used multiple and even real values, but only to establish realistic relations among node values and the magnitude of error, so even though their models were not pathological, they did not attribute the absence of pathology to real values. Sadikov *et al.* [2003] used multiple values in their analysis of king and rook versus king chess endgame. They explained the pathology, but their explanation involves multiple values

only indirectly and it is not known whether it applies to cases other than the endgame they studied.

3. Having node values distributed so that $k_i = c_b$ for all i not only simplifies calculations, it is generally agreed to be necessary [Bratko and Gams, 1982; Beal, 1982; Nau, 1982]. If $k_0 \neq c_b$ is chosen, k_i starts to oscillate between values close to 0 and 1 for relatively small i , meaning that we are dealing with games that are almost certainly won for one side and as such not interesting.
4. Most researchers [Bratko and Gams, 1982; Beal, 1982; Nau, 1982; Scheucher and Kaindl, 1998; Luštrek, 2004] agreed that similarity of positions close to each other eliminates the pathology, although they arrived at this conclusion in different ways. Pearl [1983] claimed that early terminations are the culprit, but these can also be interpreted as a form of node-value dependence.
5. Pearl [1983] showed that in order to overcome the pathology, the error of the evaluation function must decrease exponentially with the depth of search. It is generally believed that the quality of the evaluation cannot vary enough to account for the absence of pathology. Scheucher and Kaindl [1998] did use depth-dependent error. And although such error made increased depth of search more beneficial, node-value dependence was required to altogether eliminate the pathology.

The conclusion one can make based on the existing literature on the minimax pathology is that the pathology is usually not observed in real games because their position values are not independent of each other. This conclusion is reinforced by the fact that multiple authors have arrived at it in different ways.

3 A Minimax Model Based on Real Values

Even though all the explanations for the absence of pathology in minimax provided in the previous section are valid, at least under the assumptions their authors made, are these assumptions really necessary and realistic? We argue that real numbers should be used for position values, in which case another, more basic explanation is sufficient.

Both game-playing programs and humans use multivalued position evaluations. There is little doubt this is necessary in games where the final outcome is multivalued (Othello, tarok etc.). In games where the outcome can only be a loss, a win and perhaps a draw (chess, checkers etc.), multiple values might seem to be useful only as a way to express the uncertainty of a program or human. However, even given unlimited resources to determine the value of a position, in a losing position, the best one can do against a fallible and not fully known opponent is evaluate the position in terms of the probability of loss. In a winning position, even a perfect two-valued evaluation function could maintain a won position indefinitely without actually winning (or until termination due to 50-move rule in chess). In essence, multivalued evaluation function is necessary to differentiate between multiple winning (or losing, if only such are available) moves. Scheucher and Kaindl [1998] demonstrated on chess that a two-valued evaluation function performs poorly compared to a multivalued one.

We propose a minimax model similar to Beal's basic model, except that it uses real numbers for position values:

1. game tree has a uniform branching factor;
2. nodes of the tree have real values;
3. if the real node values are converted to losses and wins, they are distributed so that at each level of the tree the proportion of losses for the side to move is the same;
4. node values within each level of the tree are independent of each other;
5. the error of heuristic evaluation at a node at the lowest level of search, being normally distributed noise, is independent of the depth of search and the true value of the node.

Game trees built according to our model are assigned independent uniformly distributed values from $[0, 1]$ interval to the leaves at level $d_{\max}$. These are true values; true values of internal nodes are obtained by backing up the true leaf values using the minimax rule. When searching to depth d , heuristic values at level d are generated by corrupting the true values with normally distributed noise representing the error of the heuristic evaluation function; heuristic values of nodes at levels $< d$ are obtained by backing up the corrupted values at level d using the minimax rule.

Two types of error can be observed at the root of a game tree: *position error*, which is the absolute difference between the true and the heuristic value of the root, and *move error*, which is the probability of choosing a wrong move because of position error at the root's descendants. However, neither type of error corresponds directly to the error in two-value models used in most of the previous research: *two-value error* is defined as the probability of mistaking a loss at the root for a win or vice versa.

In order to measure two-value error, real values must be converted to losses and wins. This can be accomplished by establishing a threshold t : the values below it are considered losses and the values above it wins. According to Beal's assumption 3, if negamax representation is used, $k_i = c_b$ for all i . We do not use negamax representation in our model, so k_i alternates between c_b and $1 - c_b$. Since true values of the leaves are distributed uniformly in $[0, 1]$ interval, $k_d = c_b$ is achieved by setting $t = c_b$. Even though real-value minimaxing is used, k_i behaves as desired for $i > 0$. This happens for two reasons. First, leaf values in our real-value model, converted to two values, correspond exactly to leaf values in Beal's basic model. The probability of a loss at a leaf with value X is $P(X < t)$, which for uniform distribution in $[0, 1]$ interval and $t = c_b$ equals c_b . In Beal's basic model, the probability of a loss at each leaf is k_d , which also equals c_b . Second, real- and two-value minimaxing are equivalent in the sense that performing minimax on losses and wins from level i to $j < i$ gives the same results at level j as performing minimax on the underlying real values from level i to j and converting them to losses and wins at level j . This is illustrated in Figure 1; losses are marked with “-” and wins with “+”.

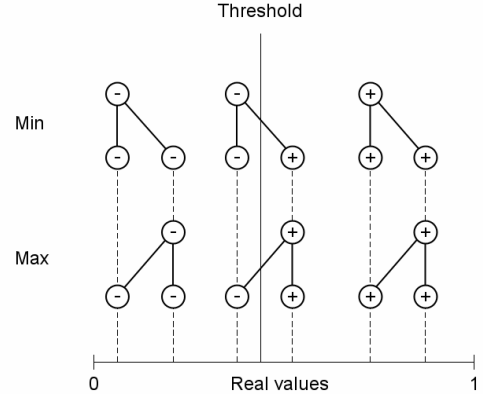


Figure 1: Equivalence of real- and two-value minimaxing.

We conducted Monte Carlo experiments with game trees generated according to our model. Only the results for $b = 2$ and $d_{\max} = 10$ are presented in this paper; the results for larger branching factors and depths are similar. The results are averaged over 10,000 game trees. For each tree, there were 10 repetitions with randomly generated noisy values for each d . Figure 1 shows position, move and two-value error at the root of the game tree with respect to the depth of search; standard deviation of noise is 0.1.

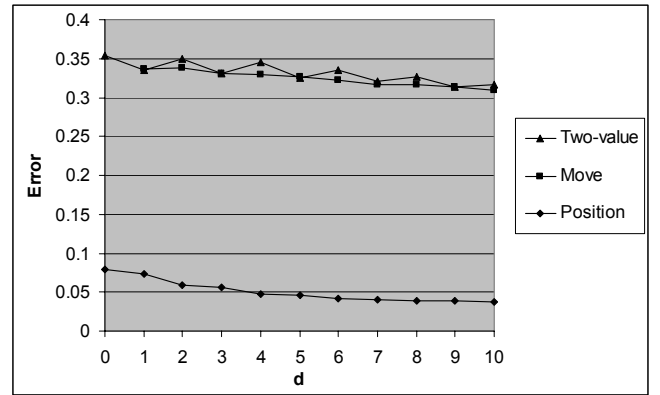


Figure 2: Error at the root with respect to the depth of search.

As can be seen in Figure 2, all three types of error decrease with the depth of search (with the exception of even/odd level fluctuations of two-value error). Note that these observed behaviors are different from Beal's original results [1980], which were pathological.

Even though our primary concern is comparison with Beal's basic model, we checked whether the absence of pathology occurs only under the described settings or is it more general. We tried uniform distribution of error and normal distribution of node values as well as different forms of dependence among node values. None of the experiments yielded pathology except for some rare cases where slightly pathological behavior was caused by very large static error. There was no pathology in terms of position error, but move and two-value error did in some cases behave pathologically when their static values were close to 0.5. However, since 0.5 is the point where evaluations become completely random, this seems to be of little practical importance.

4 Why is Our Model Not Pathological?

Considering that our model is very similar to Beal's, why is it not pathological? To answer this question, we must examine two-value error at the lowest level of search. Beal's assumption 5 states that it should be constant with search depth, but in our model, real-value position error is set to be constant instead (which is achieved by using normally distributed noise with the same standard deviation at all levels). Two-value error at the lowest level of search is shown in Figure 3; standard deviation of noise is 0.1.

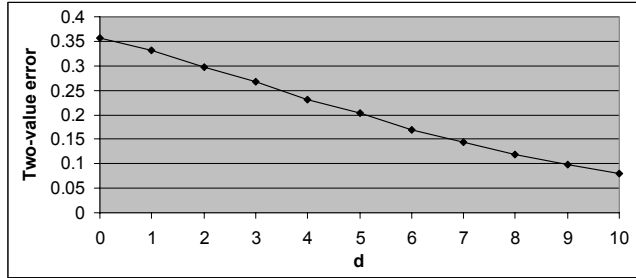


Figure 3: Two-value error at the lowest level of search with respect to the depth of search.

As can be seen in Figure 3, two-value error at the lowest level of search decreases with the depth of search. This is different from Beal's assumption 5. However, to eliminate the pathology, Pearl [1983] observed that if the error is small, it should decrease by a factor of 1.528 every two levels, i.e. exponentially with the depth of search, while in Figure 3, it decreases roughly linearly. For Pearl's observation to be true, the error should be quite small, though. For example, if the error at depth 10 is 0.1 (the value chosen by Bratko and Gams [1982] and close to the values we experimented with), Pearl's approximation gives two-value error 0.8326 at depth 0, while the exact error is 0.3932. We chose not to work with smaller errors in this paper because move and two-value error are probabilities computed from frequencies and when they are very small, very large samples are required to obtain meaningful results.

If noise introduced at the lowest level of search is adjusted so that two-value error at the lowest level of search is always 0.1, position error at the lowest level of search increases with the depth of search as shown in Figure 4.

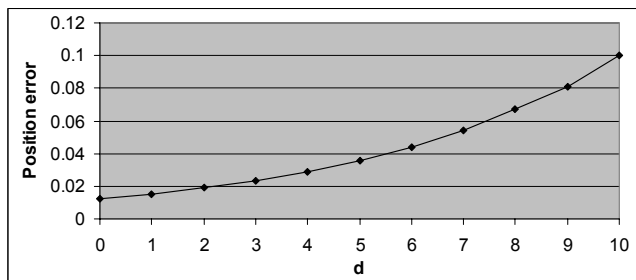


Figure 4: Position error at the lowest level of search with respect to the depth of search when two-value error at the lowest level of search is always 0.1.

Figure 5 shows two-value error at the root when two-value error at the lowest level is always 0.1; results for Beal's basic model are shown for comparison.

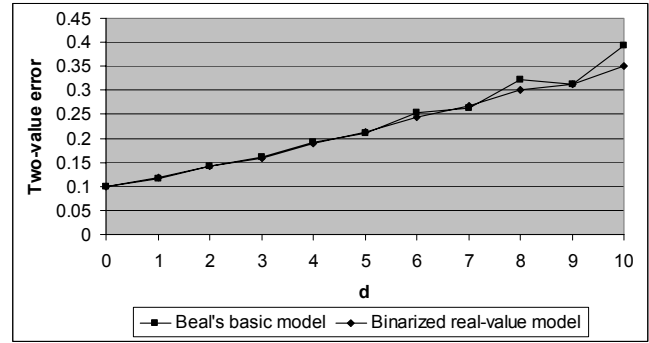


Figure 5: Two-value error at the root with respect to the depth of search when two-value error at the lowest level of search is always 0.1.

As can be seen in Figure 5, the results for binarized real-value model and the results for Beal's basic model match quite well. The matching is not perfect because in the real-value model, the probability of a false win at the lowest level of search is higher than the probability of a false loss. This happens because false wins occur in $[0, t)$ interval, while false losses occur in $(t, 0]$ interval. Since $t = c_2$, the former is smaller, therefore node values are on average closer to the threshold, and hence two-value error is more likely. The ratio of probability of a false win : probability of a false loss is $(1 - t) : t$. If the overall probability of two-value error is to remain 0.1, the appropriate settings in Beal's model are $p_d = 0.05 / t$ and $q_d = 0.05 / (1 - t)$. Under these settings, the models match perfectly.

5 Minimax Pathology in General

What remains to be considered is whether constant position error or constant two-value error at the lowest level of search is more realistic. There is no pathology in the former case, while the latter case corresponds to Beal's basic model and is pathological. Game-playing programs use real (or at least multiple) values in their evaluation functions. Position error is the most direct representation of the fallibility of these functions and there is no reason to believe that it should increase with the depth of search as shown in Figure 4. But can we expect two-value error to behave as shown in Figure 3? We will show here that the answer is "yes". Game-playing programs are generally not concerned with two-value error; if they were, one can easily imagine that it would be large in uncertain positions whose values are close to the threshold and small in clearly lost or won positions far from the threshold. If both sides are to have comparable chances to win at the root, the value at the root should be close to the threshold. Each level downwards from the root is one move away from the root position. Position values usually change gradually, so with each move, position values can be more different from the root value. Therefore the average distance of a position value from the threshold at lower levels can indeed be expected to

be larger than at higher levels, as was also stated by Scheucher and Kaindl [1998]. This is illustrated in Figure 6; darker area represents higher probability of two-value error.

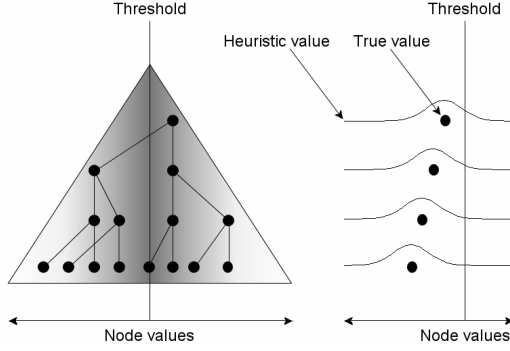


Figure 6: Distance of nodes values from the threshold and its relation to two-value error.

In game trees with independent node values, the effect of position error on two-value error can be analyzed mathematically. For simplicity, we will only consider $b = 2$ and limit node values to $[0, 1]$ interval.

Due to space limitation, we will only examine false losses. Consider the probability of a false loss at a node with true value X and heuristic value $X - e$, where e is the node's position error. False loss means that $X > t$ and the heuristic value is on the other side of the threshold, i.e. $X - e < t$. The probability for such a mistake to happen at a node whose true real value is distributed according to distribution function $F(x)$ is calculated according to equation (4).

$$P(X > t \wedge X - e < t) = P(t < X < t + e) = F(t + e) - F(t) \quad (4)$$

Consider now the probability of a false loss at different levels in a game tree. Let $F_i(x)$ be the distribution function of node values at i -th level of the game tree. If $i - 2$ is a max level, $F_{i-2}(x)$ is calculated from $F_{i-1}(x)$ according to equation (5).

$$F_{i-2}(x) = P(X_{i-2} < x) = P(X_{i-1} < x)^2 = F_{i-1}(x)^2 \quad (5)$$

If $i - 1$ is a min level, $F_{i-1}(x)$ is calculated from $F_i(x)$ according to equation (6).

$$\begin{aligned} F_{i-1}(x) &= P(X_{i-1} < x) = 1 - P(X_{i-1} > x) = \\ &= 1 - P(X_i > x)^2 = 1 - (1 - P(X_i < x))^2 = 1 - (1 - F_i(x))^2 \end{aligned} \quad (6)$$

In order to calculate $F_{i-2}(x)$ from $F_i(x)$ in one step, (5) and (6) are joined into equation (7).

$$\begin{aligned} F_{i-2}(x) &= F_{i-1}(x)^2 = (1 - (1 - F_i(x))^2)^2 = \\ &= 4F_i(x)^2 - 4F_i(x)^3 + F_i(x)^4 \end{aligned} \quad (7)$$

We will show that the probability of a static false-loss error at higher levels is usually greater than at lower levels, which is expressed by inequality (8).

$$\begin{aligned} P(\text{false loss at level } i-2) &> P(\text{false loss at level } i) \\ F_{i-2}(t+e) - F_{i-2}(t) &> F_i(t+e) - F_i(t) \end{aligned} \quad (8)$$

Inequality (8) means that the difference between the values of distribution function at points $t + e$ and t is greater at higher levels – in other words, that the distribution function is steeper at higher levels. Let us take as an

example our model from section 3: $d_{\max} = 10$ and leaf values are distributed uniformly, therefore $F_{10}(x) = x$. Equation (7) can be used to calculate $F_8(x) = 4x^2 - 4x^3 + x^4$. Figure 7 shows $F_8(x)$ and $F_{10}(x)$ in our model.

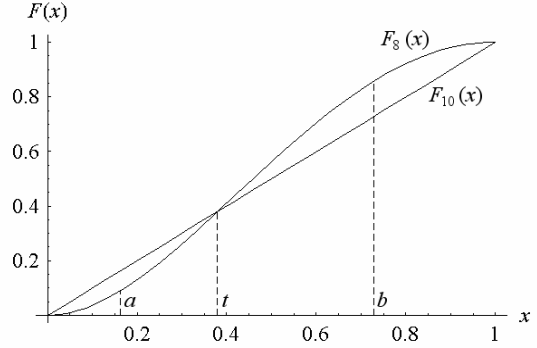


Figure 7: Distribution functions of node values at levels 8 and 10 in our model from section 3.

As can be seen in Figure 7, $F_8(x)$ is steeper than $F_{10}(x)$ between $x = a$ and $x = b$. To determine where $F_{i-2}(x)$ is steeper than $F_i(x)$ independently of $F_i(x)$, inequality (9) must be solved; note that $F_i(x)$ is written as F_i .

$$\begin{aligned} \frac{dF_{i-2}}{dF_i} &> \frac{dF_i}{dF_i} \\ 8F_i - 12F_i^2 + 4F_i^3 &> 1 \end{aligned} \quad (9)$$

The expression dF_{i-2}/dF_i as a function of F_i is shown in Figure 8.

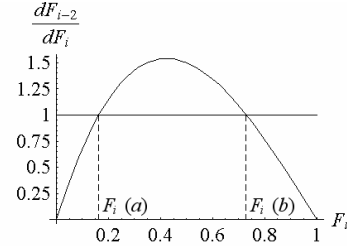


Figure 8: dF_{i-2}/dF_i as a function of F_i .

With the help of Figure 8 we can solve inequality (9): $0.1624 < F_i < 0.7304$. Values a and b in Figure 7 are therefore $a = F_{10}^{-1}(0.1624)$ and $b = F_{10}^{-1}(0.7304)$; since $F_{10}(x) = x$, this means $a = 0.1624$ and $b = 0.7304$. So whenever the values of $F_i(t+e)$ and $F_i(t)$ are in the interval between 0.1624 and 0.7304, false-loss error at level $i-2$ is greater than false-loss error at level i for any distribution function $F_i(x)$. Since $F_i(t) = k_i$, if k_i is to be the same for all i , most reasonable distribution functions should satisfy the condition $0.1624 < F_i(t) < 0.7304$.

We now see that two-value error at higher levels is smaller than at lower levels. But is it smaller enough? If p_i and q_i when searching to depth $d_{\max}$ were computed for all i , these values could be used for p_d and q_d when searching to depths $d < d_{\max}$. Under these conditions, two-value error at the root would be the same for searches to all depths – minimax would be neither pathological nor beneficial. If we can prove that given position error e causes greater two-value error at level $i-2$ if it is introduced at level $i-2$

(denoted P_1) than if it is introduced at level i and then backed-up to level $i - 2$ (denoted P_2), we will also have proven that two-value error as a result of depth-independent position error decreases from the leaves towards the root sufficiently to make minimax non-pathological. P_1 and P_2 are calculated using equations (4) and (7) in different order, resulting in equations (10) and (11).

$$\begin{aligned} P_1 &= F_{i-2}(t+e) - F_{i-2}(t) = \\ &= 4F_i(t+e)^2 - 4F_i(t+e)^3 + F_i(t+e)^4 - \\ &\quad - (4F_i(t)^2 - 4F_i(t)^3 + F_i(t)^4) \end{aligned} \quad (10)$$

$$\begin{aligned} P_2 &= 4(F_i(t+e) - F_i(t))^2 - 4(F_i(t+e) - F_i(t))^3 + \\ &\quad + (F_i(t+e) - F_i(t))^4 \end{aligned} \quad (11)$$

We solved inequality $P_1 > P_2$ on $[0, 1]$ interval using Mathematica software, resulting in (12).

$$F_i(t) \leq F_i(t+e) \leq \frac{6 + F_i(t) - \sqrt{4 + 12F_i(t) - 7F_i(t)^2}}{4} \quad (12)$$

The upper limit for $F_i(t+e)$ from inequality (12) is shown in Figure 9.

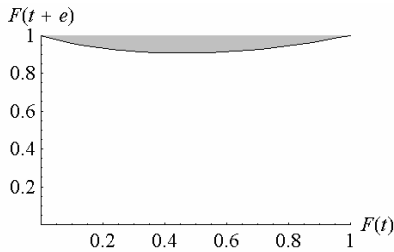


Figure 9: $F(t+e)$ as a function of $F(t)$.

As can be seen in Figure 9, minimax will behave pathologically only when $F(t+e)$ is close to 1 (shaded area). Since $F(t)$ is generally not expected to be close to 1, this means a large position error. This is consistent with our observations from section 3 where there was no pathology except in some cases when static error was very large.

6 Conclusion

To analyze the pathology, we designed a minimax model with real-number position values. The model did not behave pathologically under a wide range of settings, as long as normally (or uniformly) distributed noise used to model the error of heuristic evaluations was independent of the depth of search. However, under these settings, two-value error was not independent of the depth of search, which is contrary to what was assumed in most of the previous research. Due to minimax relations among the true values, both types of error cannot be independent of the depth of search simultaneously, because at greater depths, position values are on average farther away from the threshold separating losses from wins; therefore at greater depths, the same position error causes smaller two-value error. In real games, this is caused by position values starting close to the threshold at the root and dispersing gradually as we advance downwards through the game tree. We showed

mathematically that in game trees with independent node values, two-value error at higher levels is also smaller than at lower levels. Furthermore, we showed that the reduction of the error is in most cases sufficient to eliminate the pathology. The pathology sometimes persists, particularly when the error is very large, but such cases are probably rare. Establishing which cases exactly are these is left for further work.

In summary, the explanation for the minimax pathology, or the absence of it, presented in this paper, is a necessary consequence of a real-value minimax model, and does not require any other assumption. Even if one were to claim that in a purely theoretical sense, real values are inappropriate, they are certainly necessary in practice. And the minimax pathology is considered pathological because it is at odds with what is observed in practice.

References

- [Beal, 1980] Don F. Beal. An analysis of minimax. In *Advances in Computer Chess 2*, pages 103–109, 1980. Edinburgh University Press.
- [Beal, 1982] Don F. Beal. Benefits of minimax search. In *Advances in Computer Chess 3*, pages 17–24, 1982. Pergamon Press.
- [Bratko and Gams, 1982] Ivan Bratko and Matjaž Gams. Error analysis of the minimax principle. In *Advances in Computer Chess 3*, pages 1–15, 1982. Pergamon Press.
- [Luštrek, 2004] Mitja Luštrek. Minimax pathology and real-number minimax model. In *Proceedings of the Thirteenth International Electrotechnical and Computer Science Conference*, Volume B, pages 137–140, Portorož, Slovenia, September 2004. Slovenian Section IEEE.
- [Michon, 1983] Gerrard P. Michon. *Recursive Random Games: A probabilistic model for Perfect Information Games*. Ph.D. thesis, University of California, Los Angeles, 1983.
- [Nau, 1979] Dana S. Nau. *Quality of decision versus depth of search on game trees*. Ph.D. thesis, Duke University, 1979.
- [Nau, 1982] Dana S. Nau. An investigation of the causes of pathology in games. *Artificial Intelligence*, 19(3): 257–278, November 1982.
- [Pearl, 1983] Judea Pearl. On the nature of pathology in game searching. *Artificial Intelligence*, 20(4): 427–453, July 1983.
- [Sadikov et al., 2003] Aleksander Sadikov, Ivan Bratko, and Igor Kononenko. Search vs knowledge: Empirical study of minimax on KRK endgame. In *Advances in Computer Games: Many Games, Many Challenges*, pages 33–44, 2003. Kluwer Academic Publishers.
- [Scheucher and Kaindl, 1998] Anton Scheucher and Hermann Kaindl. Benefits of using multivalued functions for minimaxing. *Artificial Intelligence*, 99(2): 187–208, March 1998.

A New Approach to Multiobjective A* Search

L. Mandow and J.L. Pérez de la Cruz

Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga 29071 - Málaga (Spain)
{lawrence, perez}@lcc.uma.es

Abstract

The paper presents a new algorithm for multiobjective heuristic graph search problems. The algorithm presents some nice properties that are easily proven. Additionally, empirical tests show that substantial savings in memory can be achieved over previous proposals.

1 Introduction

The multiobjective search problem is an extension of the shortest path problem where arc costs become vectors. This is a genuinely different problem since vector costs induce only a partial order relation.

Loui [1983] studied the extension of Dijkstra's algorithm to the multiobjective case, and showed that some stochastic search problems can be reduced to multiobjective ones. Stewart and White [1991] outlined MOA*, a multiobjective extension of A*, and presented proofs on admissibility, node expansion, and comparison of heuristics. Dasgupta et al. [1999] extended this work in several ways, including a version for non-monotone heuristics (MOA**), with applications in the area of VLSI design and the log cutting problem. Perny and Spanjaard [2002] presented a generalization of MOA* with application to a Web access problem. Mandow and Pérez de la Cruz [2003] presented a systematic extension of the heuristic search paradigm to the multicriteria case, and admissibility conditions for algorithms with different multicriteria preference relations (multiobjective, multiattribute, lexicographic, and goal-based).

This paper reconsiders the extension of A* to the multiobjective case and presents a new algorithm. The basic operations in A* are selection and expansion of open nodes at each iteration. In A* each open node stands for a single partial solution path that can be further expanded. However, as explained in the paper, this is no longer the case in multiobjective problems, where many interesting paths may reach the same node. The MOA* algorithm [Stewart and White, 1991], and subsequent extensions, were devised preserving node selection and expansion as the algorithm's basic operations. This paper presents an extension of A* to the multiobjective case that preserves *path* selection and expansion as the basic operations. Admissibility can be proven very easily. The new algorithm also shows important properties,

analogous to those of A*, that are not shared by MOA*. Additionally, empirical tests show that the new algorithm offers substantial savings in memory over MOA*.

The paper is organized as follows. Section 2 reviews previous relevant work in scalar search and points out analogies and differences with the multiobjective search problem. Section 3 presents a new algorithm and illustrates its behaviour with a simple example. Relevant differences with MOA* are identified. Section 4 proves the admissibility of the new algorithm and discusses its efficiency. The performance results of multiobjective search over a set of randomly generated grid search problems is presented in section 5. Finally, conclusions and future research are briefly summarized.

2 Scalar and multiobjective search

2.1 Previous results in scalar graph search

The shortest path problem can be stated as follows: given a locally finite labelled directed graph $G = (N, A, c)$, of $|N|$ nodes, and $|A|$ arcs (n, n') labelled with positive costs $c(n, n') \in \mathbb{R}$, a start node $s \in N$, and a set of goal nodes $\Gamma \subseteq N$, find the minimum cost path in G from s to a node in Γ .

The A* algorithm [Hart *et al.*, 1968] is an efficient solution to this problem. It is a best-first search algorithm that uses a search tree to store the set of known interesting partial solution paths, a list of open nodes that can be further expanded, and a characteristic evaluation function $f(n) = g(n) + h(n)$ to select the next open node to be expanded. While function $g(n)$ denotes the known cost of the path stored in the tree from s to n , the heuristic function $h(n)$ estimates of the cost of a solution from node n to a goal node. Heuristic functions can be devised for particular problems depending on their semantics.

Beautiful results have been presented to analyse the relation between the accuracy of $h(n)$ and the properties of A*.

Admissibility: Let $h^*(n)$ be the real optimal cost of a path from n to a goal node. When the heuristic is an optimistic estimate ($h(n) \leq h^*(n) \forall n$), the search is admissible, i.e. if a solution exists it is guaranteed to find an optimal one. With additional constraints ($h(n) \geq 0 \forall n$, and $c(n, n') \geq \epsilon > 0 \forall (n, n') \in A$) A* is admissible even on infinite graphs.

Efficiency: When $\forall n h(n) = 0$, A* behaves like Dijkstra's algorithm. When $h(n)$ is *consistent* or satisfies the equivalent

monotone property,

$$h(n) + c(n, n') \leq h(n') \quad \forall (n, n') \in A \quad (1)$$

A* solves the problem in $O(|N|)$ iterations, storing $O(|N|)$ nodes in memory, in the worst case. Particularly, if c^* denotes the optimal solution cost, A* will always expand all nodes with $f(n) < c^*$, but at most those with $f(n) \leq c^*$. Therefore, given an optimistic estimate, big values of $h(n)$ can push the evaluation of more and more nodes beyond the $f(n) = c^*$ frontier, reducing search effort.

Optimality: When the heuristic is monotone A* can also be proven optimal among the class of admissible best-first algorithms that are guided by path-dependent evaluation functions [Dechter and Pearl, 1985].

2.2 Extension to the multiobjective case

The multiobjective search problem can be stated as follows: given a locally finite labelled directed graph $G = (N, A, \vec{c})$, of $|N|$ nodes, and $|A|$ arcs (n, n') labelled with positive vectors $\vec{c}(n, n') \in \mathbb{R}^q$, a start node $s \in N$, and a set of goal nodes $\Gamma \subseteq N$, find the set of non-dominated cost paths in G from s to nodes in Γ .

In multiobjective problems cost vectors $\vec{c}(n, n')$ induce only a partial order preference relation $\prec$ called *dominance*,

$$\forall \vec{f}, \vec{f}' \in \mathbb{R}^q \quad \vec{f} \prec \vec{f}' \Leftrightarrow \forall i \quad f_i \leq f'_i \wedge \vec{f} \neq \vec{f}' \quad (2)$$

where f_i denotes the i -th element of vector $\vec{f}$.

Therefore, given two q -dimensional vectors $\vec{f}$ and $\vec{f}'$, it is not always possible to rank one as better than the other. For example, in a two dimensional cost space, vector $(2, 3)$ dominates $(2, 4)$, but no dominance relation exists between vectors $(2, 3)$ and $(3, 2)$.

The essence of the multiobjective search problem is to find the set of *all* non-dominated solution paths. Therefore, an analogy could be traced to a version of A* that aims at finding all optimal paths. However, some important differences with the scalar search problem can be pointed out.

First of all, given the dominance preference relation, two (or more) uncomparable (non-dominated) paths may reach any given node from the same or different parents. Several important consequences can be identified:

1. The search tree used by A* to record the best known path to generated nodes is no longer sufficient. A directed acyclic graph can be used instead, to record the set of non-dominated known paths to generated nodes.
2. The number of generated nodes may no longer be a realistic estimate of the memory needed by the algorithm. All arcs and non-dominated cost vectors reaching each node need to be recorded as well. Let M be an upper bound on the size of the largest set of non-dominated vectors in a node, and $O(M)$ a bound on the size of the approximations to these sets. In the worst case $O(MN)$ cost vectors need to be stored. Bentley et al. [1978] prove that the average number of non-dominated q -dimensional vectors in a set of size L is $O((\log L)^{q-1})$, provided all $(n!)^q$ relative orderings are equally probable.

3. Finally, each time a new path is generated to a known node, its cost may need to be tested for dominance with the set of all known costs reaching the node.

In general, heuristic estimates will involve a set of vectors $H(n)$ for each node n , estimating cost vectors of paths from n to each goal node. Therefore, for each path P_{sn} from s to n with cost $\vec{g}_P$, there will be a set of heuristic evaluation vectors, $F(P_{sn})$. This function is the multiobjective analogue to $f(n)$ in A*,

$$F(P_{sn}) = F(n, \vec{g}_P) = \text{nondom}\{\vec{f} \mid \vec{f} = \vec{g}_P + \vec{h} \wedge \vec{h} \in H(n)\}$$

where $\text{nondom}(X)$ denotes the set of non-dominated vectors in set X .

Finally, note that at each iteration, A* selects and expands an open node, i.e. considers all the possible extensions of the path stored in the search tree to that node. In A* each open node stands for a single partial solution path that can be further expanded. However, in the multiobjective case, if an acyclic graph is used instead of a tree to record partial solutions, this will no longer be the case. The MOA* algorithm [Stewart and White, 1991] was devised preserving node selection and expansion as the algorithm's basic operations. The next section introduces an extension of A* to the multiobjective case that preserves path selection and expansion as the basic operations.

3 A new algorithm for multiobjective A* search

3.1 Brief description

The pseudocode of a new multiobjective A* search algorithm is shown in table 1.

In accordance to the considerations in sec 2.2, we will devise our extension of A* to the multiobjective case in the following way:

- The algorithm uses an acyclic search graph SG to record interesting partial solution paths. For each node n in SG two sets, $G_{cl}(n)$ and $G_{op}(n)$, denote the sets of non-dominated cost vectors of paths reaching n that have or have not been explored yet respectively (i.e. closed or open). Each cost vector in this sets labels one or more pointers emanating from n to its parents with that cost. Initially, s is the only node in SG .
- The algorithm keeps a list $OPEN$, of partial solution paths that can be further expanded. For each node n in SG and each nondominated cost vector $\vec{g} \in G_{op}(n)$, there will be a corresponding triple $(n, \vec{g}, F(n, \vec{g}))$ in $OPEN$. Initially, $(s, \vec{g}_s, F(s, \vec{g}_s))$ is the only triple in $OPEN$.
- At each iteration, the algorithm will consider the extension of an open triple $(n, \vec{g}, F)$ that stands for a partial solution path from s to n with cost $\vec{g}$.
- Two sets, $GOALN$ and $COSTS$, record all goal nodes reached and all non-dominated cost vectors to goal nodes respectively. Once a solution is known its cost vector can be used to discard (filter) dominated open triples.

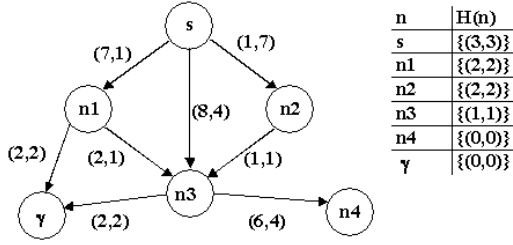


Figure 1: Sample graph and heuristic function.

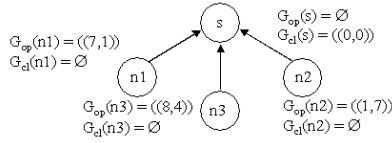


Figure 2: Search graph (iteration 2).

- Search terminates only when the *OPEN* list is empty, i.e. when all open triples have been selected or filtered.

3.2 Example

We will illustrate the workings of the previous algorithm on the sample graph in figure 1. A heuristic evaluation function $H(n)$ is also presented. A single heuristic vector is given for each node for the sake of simplicity. A trace of the search graph is shown in figures 2-6. Values of G_{op} and G_{cl} are shown at each iteration. Values that do not change from the previous iteration are omitted for brevity. Table 2 shows a trace of *OPEN*.

At iteration 1, *SG* has only node *s* at its root, $G_{op}(s) = \{(0,0)\}$, and $G_{cl}(s) = \emptyset$. The only path in *OPEN* is selected, and its three extensions to nodes *n1*, *n2*, and *n3* added to *SG* and *OPEN*. At iteration 2 two alternatives have non-dominated estimates in *OPEN* (*n1* with estimate (9, 3), and *n2* with estimate (3, 9)). Let's assume ties between non-dominated estimates are always broken choosing the smaller first component in estimated cost vectors. The path leading to *n2* would be selected, and its extension to *n3* generated. At iteration 3 two non-dominated paths lead to node *n3* (each cost vector in $G_{op}(n3)$ would label a different pointer). One of them is non-dominated in *OPEN* and selected. At iteration 4 a path leading to the goal node γ has a non-dominated estimate in *OPEN*. It is selected, added to *GOALN*, and its cost vector included in *COSTS*. At iteration 5 the path leading to node *n4* has been filtered (its estimate is dominated by the vector in *COSTS*). The only non-dominated alternatives are selected at iterations 5 and 6. At iteration 6 a new solution path leading to γ is selected. Therefore, the set *COSTS* is updated to $\{(4, 10)(9, 3)\}$. At iteration 7 all remaining alternatives are filtered and *OPEN* is empty. The algorithm would backtrack from γ returning the two paths found with costs (4, 8) and (9, 3).

Table 1: A new path expansion algorithm for multiobjective A* search.

1. CREATE:

- An acyclic search graph *SG* rooted in *s*.
- List of alternatives, $OPEN = \{(s, \vec{g}_s, F(s, \vec{g}_s))\}$.
- Two empty sets, *GOALN*, *COSTS*.

2. CHECK TERMINATION.

If *OPEN* is empty, then backtrack in *SG* from the nodes in *GOALN* and return the set of solution paths with costs in *COSTS*.

3. PATH SELECTION.

Select an alternative $(n, \vec{g}_n, F)$ from *OPEN* with $\vec{f} \in F$ non-dominated in *OPEN*, i.e.

$$\forall (n', \vec{g}_{n'}, F') \in OPEN \quad \nexists \vec{f}' \in F' \mid \vec{f}' \prec \vec{f} \quad (3)$$

Delete $(n, \vec{g}_n, F)$ from *OPEN*, and move $\vec{g}_n$ from $G_{op}(n)$ to $G_{cl}(n)$.

4. SOLUTION RECORDING.

If $n \in \Gamma$, then

- Include *n* in *GOALN* and $\vec{g}_n$ in *COSTS*.
- Eliminate from *OPEN* all alternatives $(x, \vec{g}_x, F_x)$ such that all vectors in F_x are dominated by $\vec{g}_n$ (FILTERING).
- Go back to step 2

5. PATH EXPANSION:

If $n \notin \Gamma$, then

For all successors nodes *m* of *n* that do not produce cycles in *SG* do:

- Calculate the cost of the new path found to *m*: $\vec{g}_m = \vec{g}_n + \vec{c}(n, m)$.
- If *m* is a new node
 - Calculate $F_m = F(m, \vec{g}_m)$ filtering estimates dominated by *COSTS*.
 - If F_m is not empty, put $(m, \vec{g}_m, F_m)$ in *OPEN*, and put $\vec{g}_m$ in $G_{op}(m)$ labelling a pointer to *n*.
 - Go to step 2.
- else (*m* is not a new node), in case
 - $\vec{g}_m \in G_{op}(m)$ or $\vec{g}_m \in G_{cl}(m)$: label with $\vec{g}_m$ a pointer to *n*, and go to step 2.
 - If $\vec{g}_m$ is non-dominated by any cost vectors in $G_{op}(m) \cup G_{cl}(m)$ (a path to *m* with new cost has been found), then :
 - Eliminate from $G_{op}(m)$ and $G_{cl}(m)$ vectors dominated by $\vec{g}_m$
 - Calculate $F_m = F(m, \vec{g}_m)$ filtering estimates dominated by *COSTS*.
 - If F_m is not empty, put $(m, \vec{g}_m, F_m)$ in *OPEN*, and put $\vec{g}_m$ in $G_{op}(m)$ labelling a pointer to *n*.
 - Go to step 2.
 - Otherwise: go to step 2.

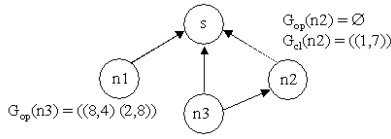


Figure 3: Search graph (iteration 3).

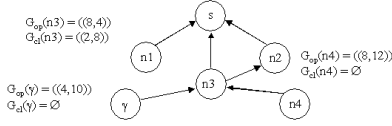


Figure 4: Search graph (iteration 4).

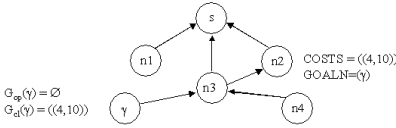


Figure 5: Search graph (iteration 5).

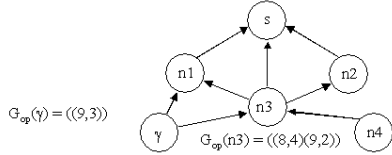


Figure 6: Search graph (iteration 6).

Table 2: *OPEN* alternatives at each iteration (dom = dominated; ← = selected).

It.	<i>OPEN</i>
1	$(s, (0, 0), ((3, 3))) \leftarrow$
2	$(n_1, (7, 1), ((9, 3)))$ $(n_2, (1, 7), ((3, 9))) \leftarrow$ $(n_3, (8, 4), ((9, 5))) \text{ dom}$
3	$(n_1, (7, 1), ((9, 3)))$ $(n_3, (8, 4), ((9, 5))) \text{ dom}$ $(n_3, (2, 8), ((3, 9))) \leftarrow$
4	$(n_1, (7, 1), ((9, 3)))$ $(n_3, (8, 4), ((9, 5))) \text{ dom}$ $(n_4, (8, 12), ((8, 12))) \text{ dom}$ $(\gamma, (4, 10), ((4, 10))) \leftarrow$
5	$(n_1, (7, 1), ((9, 3))) \leftarrow$ $(n_3, (8, 4), ((9, 5))) \text{ dom}$
6	$(n_3, (8, 4), ((9, 5))) \text{ dom}$ $(\gamma, (9, 3), ((9, 3))) \leftarrow$ $(n_3, (9, 2), ((10, 3))) \text{ dom}$

3.3 Selection and expansion: paths versus nodes

MOA* has been for more than a decade the reference multiobjective heuristic search algorithm. Unlike the algorithm presented in section 3.1, MOA* is built around the idea of node selection and expansion. Particularly, it makes no distinction in *SG* between the sets $G_{op}(n)$ and $G_{cl}(n)$, and a single set $G(n) = G_{op}(n) \cup G_{cl}(n)$ is kept for each node. A heuristic evaluation function $F(n)$ is defined for each node,

$$F(n) = \text{nondom}\{\vec{f} \mid \vec{f} = \vec{g} + \vec{h} \wedge \vec{g} \in G(n) \wedge \vec{h} \in H(n)\}$$

The *OPEN* list can be considered a list of pairs $(n, F(n))$ where nodes with non-dominated $\vec{f} \in F(n)$ are selected. The distinction between individual paths reaching each node n is lost and, accordingly, all known paths reaching each node n are either simultaneously open or closed. Essentially MOA* presents two distinguishing features when compared to the algorithm in section 3.1:

1. Each time a new non-dominated path is found to a closed node, the whole node is put back into *OPEN*.
2. Each time a node is selected for expansion, all known non-dominated paths reaching that node are extended.

The first difference may result in unnecessary reexpansion of nodes. The second may result in unnecessary extension of paths in *SG* and storage of uninteresting cost vectors. Additionally, when a goal node γ is selected, all cost vectors in $G(\gamma)$ may enter in *COSTS*. In general, it is not possible to know which of these cost vectors belong to truly non-dominated solutions until all of them have been found.

These differences can be illustrated with the example presented in section 3.2. Provided ties in the selection of *OPEN* paths were broken in the same way, search with MOA* would produce the same results in the first two iterations. In iteration 3, the selection of node n_3 for expansion would produce in MOA* the extension of *both* known non-dominated paths to that node. Accordingly, all cost vectors in $G(n_3) = \{(8, 4)(2, 8)\}$ would be extended resulting in $G(n_4) = \{(14, 8)(8, 12)\}$ and $G(\gamma) = \{(10, 6)(4, 10)\}$, and node n_3 would be marked as *closed*. Note that the new algorithm did not need to store cost vectors $(14, 8)$ in node n_4 , or $(10, 6)$ in γ .

At iteration 4 node γ would be selected, and two different cost vectors $\{(10, 6)(4, 10)\}$ would be included in *COSTS*. One of them does not belong to a truly non-dominated solution, but MOA* will not be able to tell until termination.

Finally, note that at iteration 5, the expansion of node n_1 results in a new non-dominated path to n_3 . MOA* would place accordingly n_3 back in *OPEN* with *all* its heuristic evaluation cost vectors. This implies that prior to termination, MOA* will need to reexpand n_3 . This was unnecessary in the new algorithm since the newly found path to n_3 can never lead to a non-dominated solution.

4 Properties

This section presents proofs on the admissibility of the new algorithm presented in section 3, and bounds on the number of paths selected and stored in memory.

A scalar algorithm is said to be *admissible* if it is guaranteed to return an optimal solution whenever a solution exists.

We extend the definition as follows: a multiobjective search algorithm is *N-admissible* if it terminates with the set of all non-dominated solutions whenever this set is finite and non-empty, or if it does not terminate whenever this set is infinite.

Let C^* be the set of non-dominated solution costs, $\vec{g}(P)$ the cost vector of path P , and $\preceq$ the relation “dominates or equals”. Let $\vec{m}^* = (m_1^*, m_2^*, \dots, m_q^*)$ denote the *ideal* optimal cost vector, i.e. a vector that would attain the optimal cost for each individual dimension. Note that problems where ideal optimal solutions are reachable are highly unusual.

The proofs presented in this section rely on a set of reasonable assumptions, analogous to those presented in [Stewart and White, 1991] to prove the admissibility of MOA*:

1. The graph $G = (N, A)$ to be searched is locally finite.
2. The heuristic function $H(n)$ is admissible, i.e. for all non-dominated solution paths $P^* = (s, n_1, \dots, n_i, n_{i+1} \dots \gamma_k), \gamma_k \in \Gamma$ and each subpath $P_i^* = (s, n_1, \dots, n_i)$ of P^* ,
 $\exists \vec{h} \in H(n_i) \mid \vec{g}(P_i^*) + \vec{h} \preceq \vec{g}(P^*)$
3. For infinite graphs,
 - (a) $h_k \geq 0 \quad \forall k \forall n \forall \vec{h} \in H(n)$
 - (b) $\vec{c}_k(n, n') \geq \epsilon > 0 \quad \forall k \forall (n, n') \in A$

Theorem 1 : For each non-dominated solution path $P^* = (s, n_1, \dots, n_i, n_{i+1} \dots \gamma)$ with cost $\vec{g}(P^*) = \vec{c}^*$, there is always before its discovery a subpath $P_i^* = (s, n_1, \dots, n_i)$ of P^* such that: a) P_i^* is recorded in SG ; b) $\vec{g}(P_i^*) \in G_{op}(n_i)$; c) $\exists \vec{f} \in F(P_i^*) \mid \vec{f} \preceq \vec{c}^*$.

Proof. We first consider parts (a), and (b). The proposition is true at iteration 1, when s is in *OPEN* and at the root of SG . It is also true in subsequent iterations, since subpaths of non-dominated solutions can never be pruned from SG . Let P_i^* make the proposition true for some non-dominated solution P^* . By definition, P_i^* is non-dominated to n_i , therefore $\vec{g}(P_i^*)$ will never be removed from $G_{op}(n_i)$ unless $(n_i, \vec{g}(P_i^*), F(P_i^*))$ is selected for expansion. Upon selection, if $n_i = \gamma$, then $P_i^* = P^*$ is discovered. If $n_i \neq \gamma$, a new non-dominated path to n_{i+1} , $P_{i+1}^* = (s, n_1, \dots, n_i, n_{i+1})$ will be generated, $\vec{g}(P_{i+1}^*)$ included in $G_{op}(n_{i+1})$ and P_{i+1}^* will satisfy the proposition. Part (c) follows then trivially from the definition of F and admissible heuristic. This, in turn, prevents P^* or any of its subpaths from filtering.

Theorem 2 If there is a solution, the algorithm terminates even on infinite graphs.

Proof. All best first search algorithms that prune cycles terminate on finite graphs [Pearl, 1984]. Given the assumptions for infinite graphs, all paths P with length longer than

$$\frac{\max_i \{m_i^*\}}{\epsilon}$$

will trivially attain $\vec{g}(P)$ such that $\vec{c}^* \prec \vec{f}$ for all $\vec{c}^* \in C^*$ and all $\vec{f} \in F(P)$. Since the graph G is locally finite, there can only be a finite number of partial solution paths non-dominated by C^* . Therefore, from theorem 1, all non-dominated solutions will eventually be found, and all infinite paths will be pruned or filtered from *OPEN* in a finite number of steps.

Theorem 3 The algorithm is *N-admissible*.

Proof. From theorems 1 and 2 follows that the algorithm will always terminate with all non-dominated solutions. It suffices to show that a dominated solution can never be selected. Let's assume, for the purpose of contradiction, that a dominated solution P with cost $\vec{g}_P$ is selected, i.e. exists $\vec{c}^* \in C^*$ with $\vec{c}^* \prec \vec{g}_P$, and $F(P) = \{\vec{g}_P\}$. It must be $\vec{c}^* \notin COSTS$, for otherwise P would have been filtered. Now, from theorem 1, for all $\vec{c}^* \in C^*$ not found yet there is at least one path $P_i = (s, \dots, n_i)$ such that exists $\vec{f} \in F(P_i)$ with $\vec{f} \preceq \vec{c}^*$, and $(n_i, \vec{g}(P_i), F(P_i))$ is in *OPEN*. Now, from the transitivity of the dominance relation follows that $\vec{f} \preceq \vec{c}^* \prec \vec{g}_P$ and P cannot be selected for expansion.

Corollary 1 At any time $COSTS \subseteq C^*$. Therefore, the algorithm may terminate at any time returning the set of non-dominated solutions found so far.

Corollary 2 All paths P in SG with $\vec{f} \in F(P)$ such that $\forall \vec{c}^* \in C^*, \vec{c}^* \not\prec \vec{f}$ will either be pruned at later iterations, or selected for expansion.

Corollary 3 No path P such that $\forall \vec{f} \in F(P) \exists \vec{c}^* \in C^*, \vec{c}^* \prec \vec{f}$ will ever be selected for expansion.

Corollary 4 For each path P recorded in SG such that $\forall \vec{f} \in F(P) \exists \vec{c}^* \in C^*, \vec{c}^* \prec \vec{f}$, its extensions will never be recorded in SG .

Proofs of corollaries 1, 3 and 4 are trivial from the proof of theorem 3. Corollary 2 is also trivial since otherwise the algorithm would not terminate.

Note that a version of A* that found all optimal paths would also satisfy corollary 1. Corollaries 2 and 3 give lower and upper bounds on the paths selected for expansion. These are analogous to the efficiency bounds of A* (see section 2.1).

The example presented in section 3.2 shows that corollaries 1, 3, and 4 do not apply to MOA*. Particularly, for MOA*, no upper bound can be given for the cost of paths whose extensions will be recorded in SG depending on C^* .

5 Experimental test

5.1 Search space and practical issues

Search in square grids of nodes was used to test the algorithms. These experiments provide further insight into the workings of both algorithms, as well as a preliminary evaluation of the memory savings that can be achieved with the new algorithm. In two dimensional grids, each node (identified by its coordinates) has its four neighbours as successors. Note that, for this problem, storing each node takes less space than storing each cost vector for $q > 2$.

Vector costs were generated randomly with integer components in the range $[1, 10]$. A single goal node was generated for each problem instance, at a random distance between 2 and 30 arcs from the start node. Manhattan distance to the goal can be used as an optimistic cost estimate for each component of the cost vectors.

A lexicographic order was used to sort alternatives in the *OPEN* set, as suggested in [Loui, 1983]. Particularly,

only the current lexicographically optimal evaluation vector $\vec{f}$ is placed in *OPEN* for each node with open alternatives. Therefore, the size of *OPEN* is $O(|N|)$ in the worst case for both algorithms, and alternatives can be deleted and inserted in $O(\log|N|)$ in a binary heap. Proofs on admissibility are not affected by this selection mechanism, since a lexicographic optimum is always a non-dominated vector.

5.2 Results

Table 3 summarizes relevant parameters of the search carried out over a set of 1000 problems by both algorithms¹. The set involved two-dimensional grids with three objectives. The number of nodes involved in selections was the same on both cases. The new algorithm performed 1208% more iterations, but took only 3% more time on average to complete the searches. In fact the new algorithm performed slower than MOA* only in 297 of the searches and faster in 669 of them. On the memory side, the number of nodes stored by the new algorithm was slightly higher, the number of arcs slightly lower, and the number of stored cost vectors was clearly lower. MOA* had lower requirements only in two problem instances, and needed 41.4% more space on average.

Several phenomena can be observed. The higher average size of the *OPEN* list may slow insertions and deletions in the new algorithm. An average search with MOA* needed to expand only 2.47 goal nodes to find all solution paths, while the new algorithm needed 75.57 on average (at least one for each cost vector in C^*). This suggests MOA* may benefit in some cases from earlier filtering, accounting for the slightly lower number of nodes recorded in *SG* and the two very simple cases in which MOA* stored less cost vectors. However, memory consumption is clearly dominated by the number of paths (or cost vectors) stored in *SG* where the new algorithm performs consistently better.

In summary, the new algorithm performed much better in memory requirements but found solutions slightly more slowly on average. Time values are the only measure subject to implementation details and should be taken with caution, but show clearly that the number of iterations is not a good indicator to compare the running time of both algorithms.

6 Conclusions and future work

A new admissible algorithm based on path selection and expansion has been presented for multiobjective heuristic search. The algorithm is amenable to formal analysis and presents several interesting properties: the algorithm may be stopped at any time, returning the subset of non-dominated solutions found so far; bounds can be given on the number of paths stored in *SG*, and selected for expansion, as a function of C^* . An example is shown where MOA*, a previous algorithm, violates the upper bounds. Preliminary experimental tests confirm that the new algorithm consistently saves substantial amounts of memory, though more complete experimental evaluation is needed.

The properties presented in the paper are a necessary first step in a proper analysis on the influence of monotonicity and

Table 3: Test results [average (min; max; std. dev.)].

New algorithm	
Iterations	8396.22 (6.00 ; 63722.00 ; 10024.38)
Goal nodes sel.	75.57 (1.00 ; 433.00 ; 77.26)
Nodes in <i>SG</i>	427.60 (9.00 ; 1147.00 ; 245.25)
Arcs in <i>SG</i>	805.30 (9.00 ; 2318.00 ; 498.22)
Cost vect. in <i>SG</i>	9248.19 (9.00 ; 67904.00 ; 10809.98)
Avg. <i>OPEN</i> size	171.89 (1.67 ; 524.04 ; 112.76)
Time (s)	7.99 (0.00 ; 169.18 ; 16.35)
MOA*	
Iterations	694.80 (6.00 ; 2338.00 ; 458.23)
Goal nodes sel.	2.47 (1.00 ; 15.00 ; 1.84)
Nodes in <i>SG</i>	411.11 (9.00 ; 1124.00 ; 237.66)
Arcs in <i>SG</i>	817.34 (9.00 ; 2401.00 ; 504.26)
Cost vect. in <i>SG</i>	13080.49 (9.00 ; 88684.00 ; 14648.51)
Avg. <i>OPEN</i> size	40.56 (1.67 ; 80.71 ; 16.25)
Time (s)	7.75 (0.00 ; 149.86 ; 13.66)

heuristic accuracy in the performance of the algorithm. Future work includes detailed average time complexity analysis of the algorithms, a detailed study on the influence of heuristic information on time and space efficiency, and, eventually, an extension of Dechter and Pearl's [1985] analysis on the optimality of A* to the multiobjective case.

References

- [Bentley *et al.*, 1978] J.L. Bentley, H.T. Kung, M. Schkolnick, and C.D. Thomson. On the average number of maxima in a set of vectors and applications. *Journal of the ACM*, 25(4):536–543, October 1978.
- [Dasgupta *et al.*, 1999] Pallab Dasgupta, P.P. Chakrabarti, and S.C. DeSakar. *Multiobjective Heuristic Search*. Vieweg, Braunschweig/Wiesbaden, 1999.
- [Dechter and Pearl, 1985] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, July 1985.
- [Hart *et al.*, 1968] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics SSC-4*, 2:100–107, 1968.
- [Loui, 1983] Ronald P. Loui. Optimal paths in graphs with stochastic or multidimensional weights. *Communications of the ACM*, 26(9):670–676, September 1983.
- [Madow and Pérez de la Cruz, 2003] L. Madow and J.L. Pérez de la Cruz. Multicriteria heuristic search. *European Journal of Operational Research*, 150:253–280, 2003.
- [Pearl, 1984] Judea Pearl. *Heuristics*. Addison-Wesley, Reading, Massachusetts, 1984.
- [Perny and Spanjaard, 2002] Patrice Perny and Olivier Spanjaard. On preference-based search in state space graphs. In *Proc. Eighteenth Nat. Conf. on AI*, pages 751–756. AAAI Press, July 2002.
- [Stewart and White, 1991] Bradley S. Stewart and Chelsea C. White. Multiobjective A*. *Journal of the ACM*, 38(4):775–814, October 1991.

¹Problem sets and solutions are available from the authors.

AND/OR Branch-and-Bound for Graphical Models

Radu Marinescu and Rina Dechter

School of Information and Computer Science
University of California, Irvine, CA 92697
{radum,dechter}@ics.uci.edu

Abstract

The paper presents and evaluates the power of a new framework for optimization in graphical models, based on AND/OR search spaces. The virtue of the AND/OR representation of the search space is that its size may be far smaller than that of a traditional OR representation. We develop our work on *Constraint Optimization Problems* (COP) and introduce a new generation of depth-first Branch-and-Bound algorithms that explore an AND/OR search space and use static and dynamic mini-bucket heuristics to guide the search. We focus on two optimization problems, solving Weighted CSPs (WCSP) and finding the Most Probable Explanation (MPE) in belief networks. We show that the new AND/OR approach improves considerably over the classic OR space, on a variety of benchmarks including random and real-world problems. We also demonstrate the impact of different lower bounding heuristics on Branch-and-Bound exploring AND/OR spaces.

1 Introduction

Graphical models such as constraint networks and belief networks have become a powerful representation framework for reasoning with deterministic and probabilistic information. These models use graphs to capture conditional independencies between variables, allowing a concise representation of the knowledge as well as efficient graph-based query processing algorithms.

Optimization tasks such as finding the most likely state of a belief network or finding a solution that violates the least number of constraints in a constraint network are all instances of *Constraint Optimization Problems* (COP). They are typically tackled with either *search* or *inference* algorithms. Search methods (e.g. depth-first Branch-and-Bound, best-first search) are time exponential in the number of variables and can operate in polynomial space. Inference algorithms (e.g. variable elimination, tree-clustering) are time and space exponential in a topological parameter called *tree width*. If the tree width is large, the high space complexity makes the latter methods impractical.

In this paper we focus on search. In contrast to inference algorithms, search algorithms are less sensitive to

the graph-model structure because they traverse a structure-blind search space. To overcome this problem the idea of AND/OR search spaces was introduced in the past year, and was shown to vividly display independencies encoded in the underlying graphical model [Dechter and Mateescu, 2004]. In this paper we develop *AND/OR Branch-and-Bound* (AOBB), a general algorithm for solving COPs, which explores the AND/OR search tree in a *depth-first* manner. Its efficiency depends on its guiding heuristic functions. In the past, a class of partitioning-based heuristic functions, based on the Mini-Bucket approximation and known as *static mini-bucket heuristics* was shown to be powerful for optimization problems [Kask and Dechter, 2001]. We take the idea one step further and introduce *dynamic mini-bucket heuristics*, which are computed dynamically at each node of the search tree. Both schemes are parameterized by the Mini-Bucket *i*-bound, which allows for a controllable trade-off between preprocessing and search.

We apply the AND/OR algorithms to two common optimization problems: solving Weighted CSPs [Bistarelli *et al.*, 1997] and finding the Most Probable Explanation (MPE) in belief networks [Pearl, 1988]. We experiment with both random models and real-world benchmarks. Our results show conclusively that the new AND/OR Branch-and-Bound algorithms improve dramatically over traditional OR ones, especially when the heuristic estimates are inaccurate and the algorithms rely primarily on search rather than on pruning.

2 Background

A finite *Constraint Optimization Problem* (COP) is a six-tuple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{F}, \otimes, \Downarrow, Z \rangle$, where $\mathcal{X} = \{X_1, \dots, X_n\}$ is a set of variables, $\mathcal{D} = \{D_1, \dots, D_n\}$ is a set of finite domains and $\mathcal{F} = \{f_1, \dots, f_m\}$ is a set of constraints. Constraints can be either *soft* (cost functions) or *hard* (sets of allowed tuples). Without loss of generality we assume that hard constraints are represented as (bi-valued) cost functions. Allowed and forbidden tuples have cost 0 and ∞ , respectively. The scope of function f_i , denoted $scope(f_i) \subseteq \mathcal{X}$, is the set of arguments of f_i . The operators $\otimes$ and $\Downarrow$ can be defined using the semi-ring framework [Bistarelli *et al.*, 1997], but in this paper we assume that: $\otimes_i f_i$ is a *combination* operator, $\otimes_i f_i \in \{\prod_i f_i, \sum_i f_i\}$ and $\Downarrow_Y f$ is an *elimination* operator, $\Downarrow_Y f \in \{max_{S-Y} f, min_{S-Y} f\}$, where S is the scope of function f and $Y \subseteq \mathcal{X}$. The scope of $\Downarrow_Y f$ is Y .

An optimization task is defined by $g(Z) = \downarrow_Z \otimes_{i=1}^m f_i$, where $Z \subseteq \mathcal{X}$. A *global optimization* is the task of finding the best global cost, namely $Z = \emptyset$. For simplicity we will develop our work assuming a COP instance with *summation* and *minimization* as combination and elimination operators, and a global cost function defined by $f(\mathcal{X}) = \min_{\mathcal{X}} \sum_{i=1}^m f_i$.

The *primal graph* of a COP instance has the variables $\mathcal{X}$ as its nodes and an arc connects any two variables that appear in the scope of the same function.

In practice, most complete COP solvers follow a *depth-first Branch-and-Bound* strategy which maintains an *upper bound*, the best solution cost found so far, and a *lower bound* on the optimal extension of the current assignment. Value pruning occurs as soon as the lower bound exceeds the upper bound.

3 AND/OR Search Spaces

The usual way to do search (i.e. *OR search*) is to instantiate variables in turn, following a static/dynamic linear ordering. This process defines a search tree (i.e. *OR tree*), whose nodes represent states in the space of partial assignments.

One way to exploit the independencies encoded by the graphical model is to introduce AND nodes into the search space, which will decompose the problem into separate subproblems. In the past year, [Dechter and Mateescu, 2004] introduced the concept of AND/OR search spaces for constraint networks, belief networks, and for graphical models in general. The AND/OR search space is defined using a back-bone *pseudo-tree* arrangement of the primal graph.

DEFINITION 1 (pseudo-tree) Given the primal graph G of a COP instance $\mathcal{P}$, a pseudo-tree is a rooted tree with the same set of vertices as G and has the property that adjacent vertices from G must be in the same branch of the rooted tree [Freuder and Quinn, 1985].

DEFINITION 2 (AND/OR search tree) Given a COP instance $\mathcal{P}$, its primal graph G and a pseudo-tree T of G , the associated AND/OR search tree $S_T(\mathcal{P})$ has alternating levels of OR nodes and AND nodes. The OR nodes are labeled X_i and correspond to the variables. The AND nodes are labeled $\langle X_i, a \rangle$ and correspond to value assignments in the domains of the variables. The root of the AND/OR search tree is an OR node, labeled with the root of T . The children of an OR node X_i are AND nodes labeled with assignments $\langle X_i, a \rangle$, consistent along the path from the root. The children of an AND node $\langle X_i, a \rangle$ are OR nodes labeled with the children of variable X_i in T . The path of a node $n \in S_T$, denoted $Path_{S_T}(n)$, is the path from the the root of S_T to n , and corresponds to a partial value assignment to all variables along the path.

A solution subtree Sol_{S_T} of S_T is an AND/OR subtree such that: (i) it contains the root of S_T ; (ii) if a nonterminal AND node $n \in S_T$ is in Sol_{S_T} then all of its children are in Sol_{S_T} ; (iii) if a nonterminal OR node $n \in S_T$ is in Sol_{S_T} then exactly one of its children is in Sol_{S_T} .

Example 1 Consider the graphical model in Figure 1(a) (top) describing a graph coloring problem over domains $\{0,1\}$. An AND/OR search tree based on the pseudo-tree in Figure 1(a) (bottom), and a highlighted solution subtree are given in Fig-

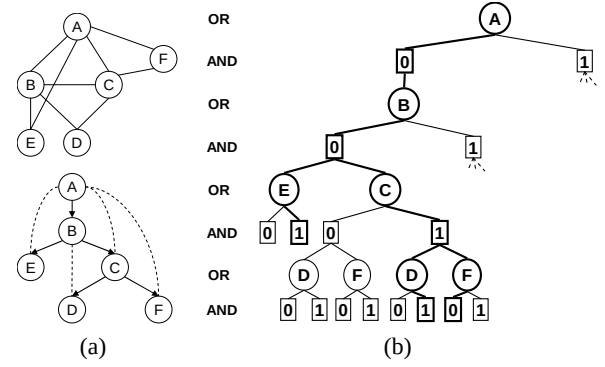


Figure 1: An AND/OR search tree.

ure 1(b). Observe that the AND node $\langle B, 0 \rangle$ ¹ roots two independent subproblems, one represented by variable $\{E\}$, and the other by variables $\{C, D, F\}$.

THEOREM 1 (Dechter and Mateescu, 2004) Given a COP instance $\mathcal{P}$ and a pseudo-tree T , its AND/OR search tree S_T is sound and complete (contains all and only solutions) and its size is $O(n \cdot \exp(m))$, where m is the depth of the pseudo-tree.

Any search algorithm that traverses the AND/OR search tree in a depth-first manner is guaranteed to have a time complexity exponential in the depth of the pseudo-tree and can operate in linear space. In contrast, the time complexity of search algorithms exploring traditional OR search trees is exponential in the number of variables. The arcs in S_T are annotated by appropriate labels of the cost functions. The nodes in S_T can be associated with a *value*, accumulating the result of the computation resulted from the subtree below.

DEFINITION 3 (label) The label $l(X_i, \langle X_i, a \rangle)$ of the arc from the OR node X_i to the AND node $\langle X_i, a \rangle$ is defined as the sum of all the cost functions values for which variable X_i is contained in their scope and whose scope is fully assigned along $Path_{S_T}(\langle X_i, a \rangle)$.

DEFINITION 4 (value) The value $v(n)$ of a node $n \in S_T$, is defined recursively as follows: (i) if $n = \langle X_i, a \rangle$ is a terminal AND node then $v(n) = l(X_i, \langle X_i, a \rangle)$; (ii) if $n = \langle X_i, a \rangle$ is an internal AND node then $v(n) = l(X_i, \langle X_i, a \rangle) + \sum_{n' \in succ(n)} v(n')$; (iii) if $n = X_i$ is an internal OR node then $v(n) = \min_{n' \in succ(n)} v(n')$, where $succ(n)$ are the children of n in S_T .

Clearly, the value of each node can be computed recursively, from leaves to root. We can show that:

PROPOSITION 1 Given an AND/OR search tree S_T of a COP instance $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{F}, +, \min \rangle$, the value function $v(n)$ is the minimal cost solution to the subproblem rooted at n , subject to the current variable instantiation along the path from root to n . If n is the root of S_T , then $v(n)$ is the minimal cost solution to $\mathcal{P}$.

Therefore, we can traverse the AND/OR search tree in a depth-first manner to compute the value of the root. This

¹In the figure we only denote the value. $\langle B, 0 \rangle$ is written as $\boxed{0}$ child of B .

approach would require linear space, storing only the current partial solution subtree. The algorithm expands alternating levels of OR and AND nodes, periodically evaluating the value function of the nodes along the current path. It terminates when the root node is evaluated with the optimal cost.

4 AND/OR Branch and Bound

If each node n at the search frontier is assigned a *heuristic lower-bound estimate* $h(n)$ of $v(n)$, then we can calculate the most promising extension of the current partial solution subtree and prune the portion of the search space that becomes irrelevant, as part of a Branch-and-Bound scheme. We call $h(n)$ a *static heuristic function*.

4.1 Lower Bounds on Partial Solution Trees

At any stage of the search, the current partial solution is a *partial solution subtree*, denoted $\mathcal{PST}$. By the nature of the search process, $\mathcal{PST}$ must be connected, must contain the root node and will have a *frontier* containing all those nodes that were generated but not yet expanded. The leaves of $\mathcal{PST}$ are called *tip nodes*. In this section we will define a *dynamic heuristic function* of a node n relative to the current $\mathcal{PST}$, which yields a more accurate lower bound on $v(n)$ than $h(n)$. For that we introduce the notions of *active path*, its *inside* and *outside contexts* and the *active partial subtree*.

DEFINITION 5 Given the current $\mathcal{PST}$, the active path $\mathcal{AP}(t)$ is the path of assignments from the root of $\mathcal{PST}$ to the current tip node t . The inside context $in(\mathcal{AP})$ of $\mathcal{AP}(t)$ contains all nodes that were fully evaluated and are children of nodes on $\mathcal{AP}(t)$. The outside context $out(\mathcal{AP})$ of $\mathcal{AP}(t)$, contains all nodes that were generated but not yet expanded and are children of the nodes on $\mathcal{AP}(t)$. The active partial subtree $\mathcal{APT}(n)$ rooted at a node $n \in \mathcal{AP}(t)$ contains the sub-path between n and t , and all OR children of the AND nodes on it.

For illustration consider the partial solution subtree in Figure 2(b) based on the pseudo-tree in Figure 2(a). The active path $\mathcal{AP}(t)$ has tip node $t = \langle E, 1 \rangle$, namely it is $(A = 1, B = 1, E = 1)$. The shaded nodes at the left of $\mathcal{AP}(t)$ are in $in(\mathcal{AP})$ and their corresponding subtrees have already been explored. $out(\mathcal{AP})$ includes the nodes $\{C, F\}$, which are also in the search frontier. The active partial subtree $\mathcal{APT}(B)$ is highlighted. It contains the nodes $\{B, \langle B, 1 \rangle, E, \langle E, 1 \rangle\}$ on $\mathcal{AP}(t)$, the OR node D from $in(\mathcal{AP})$ and the OR node F from $out(\mathcal{AP})$.

DEFINITION 6 (dynamic lower bound) Given an active partial tree $\mathcal{APT}(n)$, the dynamic heuristic evaluation function of n , $f_h(n)$, is defined recursively as follows: (i) if $\mathcal{APT}(n)$ consists only of a single node n , and if $n \in in(\mathcal{AP})$ then $f_h(n) = v(n)$ else $f_h(n) = h(n)$; (ii) if $n = \langle X_i, a \rangle$ is an AND node, having OR children $m_1, \dots, m_k$ then $f_h(n) = \max(h(n), l(X_i, \langle X_i, a \rangle) + \sum_{i=1}^k f_h(m_i))$; (iii) if $n = X_i$ is an OR node, having an AND child m , then $f_h(n) = \max(h(n), f_h(m))$.

We can prove that $f_h(n)$ is a *lower bound* on the optimal solution cost to the subproblem rooted at n , namely $f_h(n) \leq v(n)$, and also by definition $f_h(n) \geq h(n)$, indicating that the dynamic lower bound is superior to the static one.

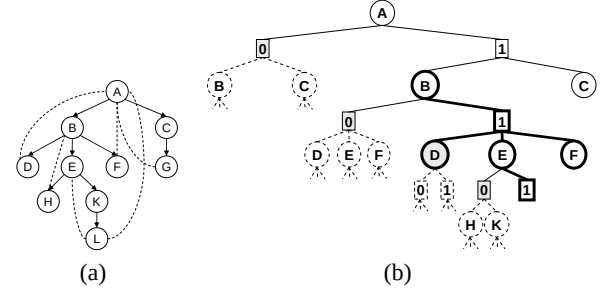


Figure 2: A partial solution subtree.

Example 2 For the active partial subtree rooted at B in Figure 2(b), the lower bound $f_h(B)$ on $v(B)$ is computed recursively as follows: $f_h(B) = \max(h(B), f_h(\langle B, 1 \rangle))$, where $f_h(\langle B, 1 \rangle) = \max(h(\langle B, 1 \rangle), l(B, \langle B, 1 \rangle) + v(D) + f_h(E) + h(F))$. Similarly, $f_h(E) = \max(h(E), f_h(\langle E, 1 \rangle)) = \max(h(E), h(\langle E, 1 \rangle))$, since $f_h(\langle E, 1 \rangle) = h(\langle E, 1 \rangle)$.

4.2 The Branch-and-Bound Procedure

In the AND/OR search space, we can calculate a *lower bound* on $v(n)$ of a node n on the active path, by using $f_h(n)$. In addition, we can compute an *upper bound* on $v(n)$, based on the portion of the search space below n that has already been explored. The upper bound $ub(n)$ on $v(n)$ is the current minimum cost solution subtree of the subproblem rooted at n .

In Figure 2(b), the upper bound on $v(B)$ is $ub(B) = \min(\infty, v(\langle B, 0 \rangle)) = v(\langle B, 0 \rangle)$, and it represents the current best cost solution rooted at B . The lower bound $f_h(B)$ on $v(B)$ is calculated as seen in the previous example. If $f_h(B) \geq ub(B)$, then searching below $t = \langle E, 1 \rangle$ of the active path is guaranteed not to reduce $ub(B)$ and therefore, the subtree rooted at $\langle E, 1 \rangle$ can be pruned.

PROPOSITION 2 (pruning rule) Given the active path $\mathcal{AP}(t)$ of a current $\mathcal{PST}$, for any node n on $\mathcal{AP}(t)$, if $f_h(n) \geq ub(n)$ then pruning the subtree below t is safe.

A depth-first AND/OR Branch-and-Bound (AOBB) algorithm that implements this pruning rule is described in Figure 3. A list called OPEN simulates the recursion stack. The list PATH maintains the current assignment (i.e. the active path). $Parent(n)$ refers to the predecessor of n in PATH, which is also the parent in the AND/OR tree, $succ$ denotes the set of successors of a node in the AND/OR tree and $ch_T(X_i)$ denotes the children of variable X_i in T . Procedure LB(n) computes the static heuristic estimate $h(n)$ of $v(n)$.

Step (3) is where the search goes forward and expands alternating levels of OR and AND nodes. Upon the expansion of n , the algorithm successively updates the *lower bound function* $f_h(m)$ for every ancestor m of n along the active path, and discontinues search below n if, for some m , $f_h(m) \geq ub(m)$.

Step (4) is where the value functions are propagated backward. This is triggered when a node has an empty set of successors and it typically happens when the node's descendants are all evaluated or when it is a dead-end. Clearly,

THEOREM 2 AOBB is sound and complete for COP.

ALGORITHM: AOBB($\mathcal{P}, T$)
Input: A COP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{F}, +, \min)$, pseudo-tree T , root X_0 .
Output: Minimal cost solution to $\mathcal{P}$.

```

(1) Initialize OPEN by adding OR node  $X_0$  to it;  $\text{PATH} \leftarrow \phi$ ;
(2) if (OPEN ==  $\phi$ )
    return  $v(X_0)$ ;
    Remove the first node  $n$  in OPEN; Add  $n$  to PATH;
    Try to prune the subtree below  $n$ :
(3) foreach  $m \in \text{PATH}$ , where  $m$  is an ancestor of  $n$ 
    if ( $f_h(m) \geq ub(m)$ )
         $v(n) \leftarrow \infty$ ; (dead-end)
        goto step (4);
    Expand  $n$  generating all its successors as follows:
     $\text{succ}(n) \leftarrow \phi$ ;
    if ( $n$  is OR node, denote  $n = X_i$ )
         $v(n) \leftarrow \infty$ ;
        foreach value  $a \in D_i$ 
             $h(\langle X_i, a \rangle) \leftarrow \text{LB}(X_i, a)$ ;
             $\text{succ}(n) \leftarrow \text{succ}(n) \cup \{\langle X_i, a \rangle\}$ ;
    else ( $n$  is AND node, denote  $n = \langle X_i, a \rangle$ )
         $A \leftarrow \{c_j \mid (X_i \in \text{var}(c_j)) \wedge (\text{var}(c_j) \subseteq \text{PATH})\}$ ;
         $v(n) \leftarrow 0$ ;  $l(X_i, \langle X_i, a \rangle) \leftarrow \sum_A c_j$ ;
        foreach variable  $Y \in \text{ch}_T(X_i)$ 
             $h(Y) \leftarrow \text{LB}(Y)$ ;
             $\text{succ}(n) \leftarrow \{Y\}$ ;
    Add  $\text{succ}(n)$  on top of OPEN;
(4) while  $\text{succ}(n) \neq \phi$ 
    if ( $n$  is OR node)
         $v(\text{Parent}(n)) \leftarrow v(\text{Parent}(n)) + v(n)$ ;
    else ( $n$  is AND node)
         $v(n) \leftarrow v(n) + l(X_i, \langle X_i, a \rangle)$ ;
         $v(\text{Parent}(n)) \leftarrow \min(v(\text{Parent}(n)), v(n))$ ;
         $\text{succ}(\text{Parent}(n)) \leftarrow \text{succ}(\text{Parent}(n)) - \{n\}$ ;
         $\text{PATH} \leftarrow \text{PATH} - \{n\}$ ;
         $n \leftarrow \text{Last}(\text{PATH})$ ;
(5) goto step (2);

```

Figure 3: AND/OR Branch-and-Bound search (AOBB).

While the time complexity of algorithm AOBB is bounded by $O(n \cdot \exp(m))$, the size of the AND/OR search space, the pruning rule is designed to yield a far better complexity.

5 Specific Lower Bound Heuristics

In this section we describe briefly two general schemes for generating static heuristic estimates $h(n)$, based on the Mini-Bucket approximation. These schemes are parameterized by the Mini-Bucket i -bound, thus allowing for a controllable trade-off between heuristic strength and its overhead. We also mention a third scheme which is based on the notion of *directional arc-consistency* and is specific to the WCSP model.

5.1 The Mini-Bucket Heuristics

Mini-Bucket Elimination (MBE) [Dechter and Rish, 2003] is an approximation algorithm designed to avoid the high time and space complexity of *Bucket Elimination* (BE) [Dechter, 1999], by partitioning large buckets into smaller subsets, called *mini buckets*, each containing at most i (called i -bound) distinct variables. The mini-buckets are then processed separately. The algorithm outputs not only a bound on the optimal solution cost, but also the collection of augmented buckets, which form the basis for the heuristics generated. The complexity is time and space $O(\exp(i))$.

Static Mini-Bucket Heuristics In the past, [Kask and Dechter, 2001] showed that the intermediate functions generated by the Mini-Bucket algorithm MBE(i) can be used to compute a heuristic function, that underestimates the minimal extension of the current assignment in a regular OR search

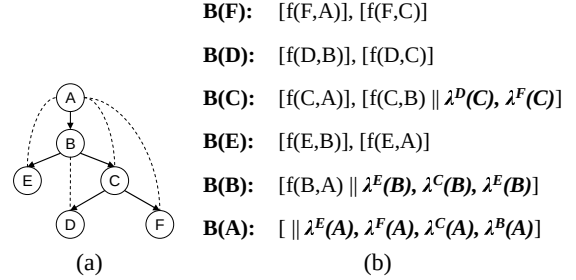


Figure 4: Schematic execution of MBE(2).

tree. In this paper we extend the idea to AND/OR search spaces as well. For that, assume that a COP instance $\mathcal{P}$ with pseudo-tree T is being solved by AOBB search, where the active path ends with some OR node X_j . Consider also the *augmented* bucket structure $\{B(X_1), \dots, B(X_n)\}$ of $\mathcal{P}$, constructed along the ordering resulted from a DFS traversal of T . For each possible value assignment $X_j = x_j$, the *static mini-bucket heuristic estimate* $h(x_j)$ of the minimal cost solution rooted at X_j can be computed as the sum of the original functions in bucket $B(X_j)$ and the intermediate functions λ^k that were generated in buckets $B(X_k)$, where X_k is a descendant of X_j in T (more details in [Kask and Dechter, 2001]).

Dynamic Mini-Bucket Heuristics This idea can be pushed one step further. Rather than pre-compiling the mini-bucket heuristic information, it is possible to generate it dynamically, during search. Therefore, the *dynamic mini-bucket heuristic* computes a lower bound by the Mini-Bucket algorithm MBE(i), at each node n in the search space, restricted to the subproblem rooted at n and subject to the current partial instantiation. Specifically, $h(x_j)$ is calculated as the sum of the original and λ^k functions residing in bucket $B(X_j)$, where λ^k 's are computed in buckets $B(X_k)$ of X_j 's descendants in T , conditioned on the current assignment of the active path.

Example 3 Figure 4(b) shows the *augmented bucket structure* generated by MBE($i=2$) for the binary COP instance displayed in Figure 4(a), along the ordering (A, B, E, C, D, F) ; square brackets denote the choice of partitioning. Assume that during search, the active path of the current partial solution subtree is $(A = a, B = b)$ and the tip node is the OR node C . The static mini-bucket heuristic estimate $h(C = c) = f(c, a) + f(c, b) + \lambda^F(a) + \lambda^F(c) + \lambda^D(c) + \lambda^D(b)$. The dynamic mini-bucket heuristic estimate $h(C = c)$, involves the same λ functions generated in buckets $B(F)$ and $B(D)$, only that the λ 's are now computed dynamically, conditioned on the current partial assignment $(A = a, B = b)$.

5.2 Directional Arc-Consistency Heuristics

We also adapted for the AND/OR search space two other successful heuristics generators, *reversible DAC counts* (RDAC) [Mesequer et al., 1999] and *maintaining full DAC* (MFDAC) [Larrosa and Schiex, 2003], which proved powerful for solving binary Weighted CSPs (details omitted for space reasons).

6 Experiments

In this section we evaluate empirically the performance of our new AND/OR Branch-and-Bound approach on two classes of

Network (n,d,c,t) [w*,h] connectivity	s-AOMB s-BBMB d-AOMB d-BBMB i=2	s-AOMB s-BBMB d-AOMB d-BBMB i=4	s-AOMB s-BBMB d-AOMB d-BBMB i=6	s-AOMB s-BBMB d-AOMB d-BBMB i=8	AORDAC BBRDAC AOMFDAC BBMFDAC
time nodes	time nodes	time nodes	time nodes	time nodes	time nodes
(20,5,100,0.7) [12,15] medium	- 2.4M - 6.3M 74.43 90K 124.45 523K	115.57 2.5M 179.34 7.6M 51.19 6.9K 80.79 34K	15.49 408K 127.77 6.5M 104.08 751 140.10 3.3K	6.91 126K 45.76 2.2M 169.30 101 176.33 537	21.70 441K 24.74 1.3M 9.99 20K 8.75 21.3K
(30,5,90,0.7) [11,16] low	- 2.8M - 6M 72.94 82K 104.15 348K	105.77 2.2M 180.00 7.5M 31.85 4.3K 69.49 22K	9.84 262K 68.90 3.3M 63.07 548 90.70 1.7K	2.71 28K 5.64 230K 94.95 111 123.42 265	78.50 1.3M 78.24 4.2M 10.72 22.2K 8.79 31.4K
(50,5,80,0.7) [8,16] sparse	67.53 1.4M - 5M 7.91 18.6K 63.50 295K	1.80 57K 145.39 4.3M 1.17 430 1.83 1K	0.12 2.6K 24.47 804K 0.95 87 0.94 112	0.62 356 1.58 36K 2.31 60 2.34 69	28.90 691K 155.58 3.9M 1.48 8.5K 1.49 14K

Table 1: Results on random binary WCSP instances.

optimization problems: Weighted CSP (WCSP) and the Most Probable Explanation (MPE) problem in belief networks².

Weighted CSP [Bistarelli *et al.*, 1997; Larrosa and Schiex, 2003] extends the classic CSP formalism with so-called *soft constraints* which assign positive integer costs to forbidden tuples (allowed tuples have cost 0). The goal is to find a complete assignment with minimum aggregated cost. The model has numerous applications in domains such as *resource allocation*, *combinatorial auctions* or *bioinformatics*.

Belief Networks [Pearl, 1988] provide a formalism for reasoning under conditions of uncertainty. A belief network represents a joint probability distribution over the variables of interest. A function of the graphical model encodes *conditional probability distribution* of a variable given its parents in the graph (also viewed as a cost function where each tuple has associated a real cost between 0 and 1). The MPE problem is the task of finding a complete assignment with maximum probability that is consistent with the evidence. It appears in applications such as *speech recognition* or *medical diagnosis*.

The *pseudo-tree* was created as suggested in [Bayardo and Miranker, 1995], by a DFS traversal of the induced graph. The latter was computed using the *min-fill* heuristic. All competing algorithms were restricted to a static variable ordering resulted from a DFS traversal of the pseudo-tree. We report the average effort, as CPU time (in seconds) and number of visited nodes (AND nodes only for the AND/OR algorithms), required for proving optimality of the solution. For all test instances we record the number of variables (n), domain size (d), number of functions (c), induced width (w*) and height of the pseudo-tree (h). A "-" indicates that a time limit was exceeded by the respective algorithm. The best results are highlighted.

6.1 Weighted CSP

For this domain we experimented with random binary WCSP problems as well as real-world benchmarks. We consider four classes of AND/OR Branch-and-Bound (AOBB) algorithms, each using a specific heuristics generator, as follows. Classes *s-AOMB/d-AOMB* are guided by static/dynamic mini-bucket heuristics, AORDAC uses RDAC based heuristics, and AOMFDAC maintains full DAC. For comparison, we include results obtained with the classic OR version of

²All our experiments were done on a 2.4GHz Pentium IV with 1GB of RAM, running Windows XP

Network	(n,d,c,w*,h)	BBMFDAC		AOMFDAC	
		time (sec)	nodes	time (sec)	nodes
CELAR6-SUB0	(16,44,57,7,10)	2.78	1,871	1.98	435
CELAR6-SUB1	(14,44,75,9,11)	2420.93	364,986	981.98	180,784
CELAR6-SUB2	(16,44,89,10,13)	8801.12	19,544,182	1138.87	175,377
CELAR6-SUB3	(18,44,106,10,13)	38889.20	91,168,896	4028.59	846,986
CELAR6-SUB4	(22,44,131,11,16)	84478.40	6,955,039	47115.40	4,643,229

Table 2: Results on CELAR6 subinstances.

each class of algorithms, denoted here by *s-BBMB*, *d-BBMB*, *BBRDAC* and *BBMFDAC*, respectively.

Random Binary Networks

Our random binary WCSP class is characterized by a five parameter model $\langle n, d, c, t, w \rangle$ [Larrosa and Schiex, 2003], where n is the number of variables, d the domain size, c the number of constraints, and t the constraint *tightness* defined as the ratio of forbidden tuples. The costs of inconsistent tuples are uniformly randomly distributed between 1 and w .

Table 1 shows results for experiments with three problem classes containing instances with medium, low and very low connectivity. We chose a maximum penalty cost w of 10 and set the constraint tightness to 70% in order to obtain over-constrained problems. For each problem class we generated 20 instances and the time limit was set to 180 seconds. The columns are indexed by the i -bound of the mini-bucket heuristics. When comparing AND/OR versus OR algorithms we notice a considerable improvement in terms of CPU time and number of nodes visited, especially for problems with low and very low connectivity. This observation verifies the theory because a relatively sparse problem is likely to produce a shallow pseudo-tree, which in turn enhances the performance of the AND/OR algorithms. In terms of the quality of the heuristics, we also observe that the static mini-buckets with a relatively large i -bound represent the best choice. However, if large i -bounds are not possible, dynamic mini-buckets with small i -bounds are preferred, especially for sparse problems. For medium and low connected problems, MFDAC proves to be cost effective with respect to the other heuristics generators.

Radio Link Frequency Assignment Problem (RLFAP)

RLFAP is a communication problem where the goal is to assign frequencies to a set of radio links in such a way that all links may operate together without noticeable interferences [Cabon *et al.*, 1999]. It can be naturally casted as a binary WCSP where each forbidden tuple has an associated penalty cost. Table 2 compares algorithms BBMFDAC and AOMFDAC for solving 5 publicly available RLFAP subinstances called CELAR6-SUB _{i} ($i = 0, \dots, 4$). We can see that the AND/OR approach is beneficial for this domain as well. In CELAR6-SUB₄, the hardest instance, AOMFDAC causes a CPU speed up of 1.8, whereas in CELAR6-SUB₃ the speed up is as much as 9.6. We also compared AOMFDAC against BTD, a recent algorithm introduced in [Jegou and Terrioux, 2004]. BTD solves the MAX-CSP version of CELAR6-SUB₄ (i.e. 0/1 penalty costs) in about 123,000 sec., whereas AOMFDAC proves optimality in only 2,574 sec. The performance of the mini-bucket based algorithms was quite poor on this domain, due to the very low quality of the heuristic estimates resulted from approximating subproblems with very large domains (up to 44 values).

Network (n,d,w*,h)	s-AOMB s-BBMB d-AOMB d-BBMB i=2		s-AOMB s-BBMB d-AOMB d-BBMB i=3		s-AOMB s-BBMB d-AOMB d-BBMB i=4		s-AOMB s-BBMB d-AOMB d-BBMB i=5	
	time	nodes	time	nodes	time	nodes	time	nodes
Mildew (35,100,4,15)	4.58 - 86.38 162.61	55K 26M 15K 83K	0.50 233.86 34.45 41.92	5,465 6.9M 1,424 5,113	0.28 0.47 1.69 1.78	35 4,970 35 363	0.66 0.81 3.02 3.13	35 4,940 35 363
Barley (48,67,7,17)	- - - -	8.5M 16M 79K 2.2M	- - 135.97 -	7.6M 18M 23K 1K	46.22 - 12.55 346.08	807K 17M 667 76K	0.56 - 45.95 -	9.6K 14M 567 86K
Munin1 (189,21,11,24)	57.36 - 66.56 -	1.2M 8.5M 185K 405K	12.08 - 12.47 -	260K 9.2M 8.1K 430K	7.20 - 10.30 -	172K 9.9M 1.6K 235K	1.66 - 11.99 14.63	43K 8.3K 523 917
Munin2 (1003,21,7,35)	- - - -	5.6M 1.7M 3.2M 33K	- - - -	6.7M 2.7M 1.2M 131K	116.19 - 524.54 -	1.8M 430K 299K 101K	1.64 - 39.22 -	21K 428K 3.8K 20K
Munin3 (1044,21,7,25)	- - - -	5.9M 1.4M 2.4M 33K	- - 68.64 -	4.9M 1.2M 58K 125K	1.31 - 3.59 -	17K 316K 5.9K 52K	0.45 - 2.84 -	6.2K 1.5M 3.8K 31K
Pigs (441,3,10,27)	- - - -	15M 2.8M 6.5M 2.2M	0.02 0.63 0.08 1.03	441 4.2K 441 614	0.02 0.63 0.08 1.08	441 4K 441 614	0.02 0.61 0.11 1.11	441 4K 441 614

Table 3: Results on Bayesian Network Repository.

6.2 Belief Networks

For the MPE domain we also experimented with random networks and real-world benchmarks, but we only report on the latter due to space limitations. We compare the algorithms using the mini-bucket based heuristics generators, namely *s-AOMB*, *d-AOMB*, *s-BBMB* and *d-BBMB*. Notice that *s-BBMB* is currently one of the best performing complete algorithms for this domain [Kask and Dechter, 2001]

Table 3 summarizes the results for experiments on 6 real-world belief networks from the Bayesian Network Repository³. The time limit was set to 600 seconds. We observe again a considerable improvement of the new AND/OR algorithms over the corresponding OR ones. If we look, for example, at the Mildew instance, *s-AOMB*(3) causes a CPU speedup of 468 over *s-BBMB*(3). In conclusion, *d-AOMB* is superior for relatively small *i*-bounds (e.g. Barley, Munin3), whereas *s-AOMB* dominates for larger *i*-bounds.

7 Conclusion

The paper investigates the impact of the AND/OR search space for graphical models on optimization tasks. We introduce a general AND/OR Branch-and-Bound algorithm and specialize it with several schemes for generating heuristic estimates that can guide the search. We focus on two common optimization problems, WCSP and Bayesian MPE, and show empirically that the new AND/OR algorithms improve dramatically over traditional OR ones on a variety of benchmarks including random and real-world problems.

Our approach leaves room for future improvements. For instance, it can be modified to traverse an AND/OR graph, rather than a tree, via caching. We should consider the effect of dynamic variable ordering. Also, we used a rather simple scheme of generating pseudo-tree arrangements, probably having non-optimal height.

³<http://www.cs.huji.ac.il/labs/compbio/Repository>

Related Work: AOBB is related to the Branch-and-Bound method proposed by [Kanal and Kumar, 1988] for acyclic AND/OR graphs and game trees. More recently, [Larrosa *et al.*, 2002] extended pseudo-tree search [Freuder and Quinn, 1985] to optimization tasks in order to boost the Russian Doll search for solving Weighted CSPs. The optimization method developed in [Jegou and Terrioux, 2004] can also be interpreted as an AND/OR search graph algorithm, however it is not a linear space algorithm.

Acknowledgments

This work was supported in part by the NSF grant IIS-0412854 and the MURI ONR award N00014-00-1-0617.

References

- [Bayardo and Miranker, 1995] R. Bayardo and D. Miranker. On the space-time trade-off in solving constraint satisfaction problems. *Proc. of IJCAI'95*, 1995.
- [Bistarelli *et al.*, 1997] S. Bistarelli, U. Montanari, and F. Rossi. Semiring based constraint solving and optimization. *Journal of ACM*, 44(2):309–315, 1997.
- [Cabon *et al.*, 1999] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. Warners. Radio link frequency assignment. *Constraints*, (4):79–89, 1999.
- [Dechter and Mateescu, 2004] R. Dechter and R. Mateescu. Mixtures of deterministic-probabilistic networks. *Proc. of UAI'04*, 2004.
- [Dechter and Rish, 2003] R. Dechter and I. Rish. Mini-buckets: A general scheme for approximating inference. *Journal of ACM*, 2003.
- [Dechter, 1999] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.
- [Freuder and Quinn, 1985] E. Freuder and M. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. *Proc. of IJCAI'85*, 1985.
- [Jegou and Terrioux, 2004] P. Jegou and C. Terrioux. Decomposition and good recording for solving max-csps. *Proc. of ECAI'04*, 2004.
- [Kanal and Kumar, 1988] L. Kanal and V. Kumar. *Search in artificial intelligence*. Springer-Verlag., 1988.
- [Kask and Dechter, 2001] K. Kask and R. Dechter. A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence*, 2001.
- [Larrosa and Schiex, 2003] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted csp. *Proc. of IJCAI'03*, pages 631–637, 2003.
- [Larrosa *et al.*, 2002] J. Larrosa, P. Meseguer, and M. Sanchez. Pseudo-tree search with soft constraints. *Proc. of ECAI'02*, 2002.
- [Meseguer *et al.*, 1999] P. Meseguer, J. Larrosa, and T. Schiex. Maintaining reversible dac for max-csp. *Artificial Intelligence*, 107(1):149–163, 1999.
- [Pearl, 1988] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan-Kaufmann, 1988.

AND/OR Cutset Conditioning

Robert Mateescu and Rina Dechter

School of Information and Computer Science

University of California, Irvine, CA 92697

{mateescu, dechter}@ics.uci.edu

Abstract

Cutset conditioning is one of the methods of solving reasoning tasks for graphical models, especially when space restrictions make inference (e.g., jointree-clustering) algorithms infeasible. The *w-cutset* is a natural extension of the method to a hybrid algorithm that performs search on the conditioning variables and inference on the remaining problems of induced width bounded by w . This paper takes a fresh look at these methods through the spectrum of AND/OR search spaces for graphical models. The resulting *AND/OR cutset method* is a strict improvement over the traditional one, often by exponential amounts.

1 Introduction

Graphical models are a widely used knowledge representation framework that captures independencies in the data and allows for a concise representation. The complexity of a reasoning task over a graphical model depends on the induced width of the graph. For inference-type algorithms, the space complexity is exponential in the induced width in the worst case, which often makes them infeasible for large and densely connected problems. In such cases, space can be traded at the expense of time by conditioning (assigning values to variables). Search algorithms perform conditioning on all the variables. Cycle-cutset schemes [Pearl, 1988; Dechter, 1990] only condition on a subset of variables such that the remaining network is singly connected and can be solved by inference tree algorithms. The more recent hybrid *w-cutset* scheme [Rish and Dechter, 2000; Bidyuk and Dechter, 2003] conditions on a subset of variables such that, when removed, the remaining network has induced width w or less, and can be solved by a variable elimination [Dechter, 1999] type algorithm.

The *AND/OR search space* for graphical models [Dechter and Mateescu, 2004] is a newly introduced framework for search that is sensitive to the independencies in the model, often resulting in exponentially reduced complexities. The traditional way of doing search consists of instantiating the n variables of the problem as if they were all connected in a chain, which results in a search tree exponential in n . In contrast, AND/OR search is based on a pseudo tree which

expresses independencies between variables, resulting in a search tree exponential in the depth m of the pseudo tree, where clearly $m \leq n$.

This paper applies the AND/OR paradigm to the cycle cutset method. We show that the *AND/OR cycle cutset* is a strict improvement of the traditional cycle cutset method (and the same holds for the extended *w-cutset* version). The result goes beyond the simple organization of the traditional cutset in an AND/OR pseudo tree, which would be just the straightforward improvement. The complexity of exploring the traditional cutset is time exponential in the number of nodes in the cutset, and therefore it calls for finding a minimal cardinality cutset $\mathcal{C}$. The complexity of exploring the AND/OR cutset is time exponential in its depth, and therefore it calls for finding a minimal depth *AND/OR cutset* $AO\text{-}\mathcal{C}$. That is, a set of nodes that can be organized in a start pseudo tree of minimal depth. So, while the cardinality of the optimal AND/OR cutset, $|AO\text{-}\mathcal{C}|$, may be far larger than that of the optimal traditional cutset, $|\mathcal{C}|$, the depth of $AO\text{-}\mathcal{C}$ is always smaller than or equal to $|\mathcal{C}|$.

2 Preliminaries

Reasoning graphical models A reasoning graphical model is a triplet $\mathcal{R} = (X, D, F)$ where X is a set of variables, $X = \{X_1, \dots, X_n\}$, $D = \{D_1, \dots, D_n\}$ is the set of their respective finite domains and $F = \{F_1, \dots, F_t\}$ is a set of real-valued functions, defined over subsets of X . The *primal graph* of a reasoning problem has a node for each variable, and any two variables appearing in the same function's scope are connected. The *scope* of a function is its set of arguments.

Belief networks A belief network can be viewed as an instance of a reasoning graphical model. In this case the set of functions F is denoted by $P = \{P_1, \dots, P_n\}$ and represents a set of conditional probability tables (CPTs): $P_i = P(X_i | pa_i)$, where pa_i are the parents of X_i . The associated directed graph G , drawn by pointing arrows from parents to children, should be acyclic. The belief network represents a probability distribution over X having the product form $P_B(\bar{x}) = P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | x_{pa_i})$. The *moral graph* of a directed graph is the undirected graph obtained by connecting the parent nodes of each variable and eliminating direction.

Constraint networks A constraint network can also be viewed as an instance of a reasoning graphical model. In this

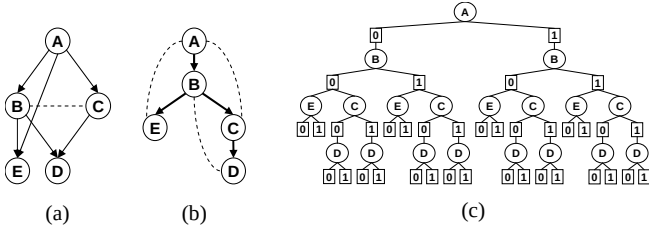


Figure 1: AND/OR Search Tree

case the functions are denoted by $C = \{C_1, \dots, C_t\}$, and the constraint network is denoted by $\mathcal{R} = (X, D, C)$. Each constraint is a pair $C_i = (S_i, R_i)$, where $S_i \subseteq X$ is the scope of the relation R_i that denotes the allowed combinations of values.

Induced-graphs and induced width An ordered graph is a pair (G, d) , where G is an undirected graph, and $d = X_1, \dots, X_n$ is an ordering of the nodes. The width of a node in an ordered graph is the number of the node's neighbors that precede it in the ordering. The width of an ordering d , denoted $w(d)$, is the maximum width over all nodes. The induced width of an ordered graph, $w^*(d)$, is the width of the induced ordered graph obtained as follows: nodes are processed from last to first; when node X is processed, all its preceding neighbors are connected. The induced width of a graph, w^* , is the minimal induced width over all its orderings. The tree-width of a graph is the minimal induced width. The path-width pw^* of a graph is the tree-width over the restricted class of orderings that correspond to chain decompositions.

3 AND/OR Search Spaces for Graphical Models

This section introduces the basics of AND/OR search spaces for graphical models as presented in [Dechter and Mateescu, 2004]. Given a graphical model $\mathcal{R} = (X, D, F)$, its AND/OR search space is driven by a pseudo tree [Freuder and Quinn, 1985]:

DEFINITION 1 (pseudo tree) Given an undirected graph $G = (V, E)$, a directed rooted tree $T = (V, E')$ defined on all its nodes is called pseudo tree if any arc of G which is not included in E' is a back-arc, namely it connects a node to an ancestor in T .

3.1 AND/OR Search Tree

Given a graphical model $\mathcal{R} = (X, D, F)$, its primal graph G and a pseudo tree T of G , the associated AND/OR search tree, denoted $S_T(\mathcal{R})$, has alternating levels of AND and OR nodes. The OR nodes are labeled X_i and correspond to the variables. The AND nodes are labeled $\langle X_i, x_i \rangle$ and correspond to the value assignments in the domains of the variables. The structure of the AND/OR search tree is based on the underlying backbone tree T . The root of the AND/OR search tree is an OR node labeled with the root of T .

The children of an OR node X_i are AND nodes labeled with assignments $\langle X_i, x_i \rangle$ that are consistent with the assignments along the path from the root, $path(x_i) =$

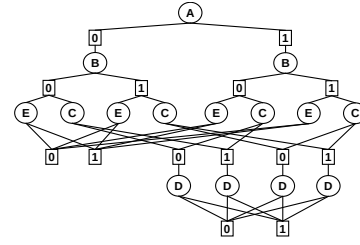


Figure 2: Context-minimal AND/OR Search Graph

$(\langle X_1, x_1 \rangle, \langle X_2, x_2 \rangle, \dots, \langle X_{i-1}, x_{i-1} \rangle)$. Consistency is well defined for constraint networks. For probabilistic networks, consistent tuples have non zero probability.

The children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of variable X_i in the pseudo tree T .

Example 1 Figure 1a shows a belief network. Figure 1b shows a pseudo tree of the moral graph, together with the back-arcs (dotted lines). Figure 1c shows the AND/OR search tree based on the pseudo tree, for binary valued variables.

The AND/OR Search Tree can be traversed by a depth first search algorithm. The arcs from X_i to $\langle X_i, x_i \rangle$ are associated with appropriate labels of the functions in F . The algorithm maintains values for each node, accumulating the result of the computation performed in the subtree below. The computation is dictated by the graphical model and task at hand, for example for belief networks, AND nodes are associated with multiplication (of the values of the independent subproblems) and OR nodes are associated with summation (over all the values of the variable). Based on earlier work [Freuder and Quinn, 1985; Bayardo and Miranker, 1996; Darwiche, 2001], it can be shown that:

THEOREM 1 Given a graphical model $\mathcal{R}$ and a pseudo tree T of depth m , the size of the AND/OR search tree based on T is $O(n \cdot \exp(m))$. A graphical model of tree-width w^* has an AND/OR search tree of size $O(\exp(w^* \cdot \log n))$.

3.2 AND/OR Search Graph

The AND/OR search tree may contain nodes that root identical subtrees. These are called *unifiable*. When unifiable nodes are merged, the search space becomes a graph. Its size becomes smaller at the expense of using additional memory. In this way, the depth first search algorithm can be modified to cache previously computed results, and retrieve them when the same nodes are encountered again. Some unifiable nodes can be identified based on their contexts [Darwiche, 2001]. The context of an AND node $\langle X_i, x_i \rangle$ is defined as the set of ancestors of X_i in the pseudo tree, including X_i , that are connected to descendants of X_i . It is easy to verify that the context of X_i d-separates [Pearl, 1988] the subproblem below X_i from the rest of the network. The context minimal AND/OR graph is obtained by merging all the context unifiable AND nodes.

Example 2 Figure 2 shows the context-minimal AND/OR search graph of the problem and pseudo tree from Figure 1.

It can be shown that [Bayardo and Miranker, 1996; Dechter and Mateescu, 2004]:

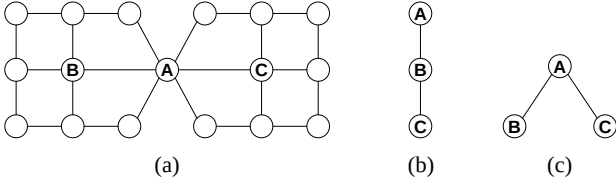


Figure 3: Traditional cycle cutset viewed as AND/OR tree

THEOREM 2 Given a graphical model $\mathcal{R}$, its primal graph G and a pseudo tree T , the size of the context minimal AND/OR search graph based on T is $O(\exp(w_T^*(G)))$, where $w_T^*(G)$ is the induced width of G (extended with the pseudo tree extra arcs) over the ordering given by the depth first traversal of T .

4 Cycle Cutset Explored by AND/OR Search

The AND/OR paradigm exploits the problem structure, by solving independent components separately. This fundamental idea can also be applied to the cycle cutset method, or reasoning by conditioning [Pearl, 1988].

DEFINITION 2 (cycle cutset) Given a graphical model $\mathcal{R} = (X, D, F)$, a cycle cutset is a subset $\mathcal{C} \subset X$ such that the primal graph of $\mathcal{R}$ becomes singly connected if all the nodes in $\mathcal{C}$ are removed from it. An optimal cycle cutset is one having the minimum number of variables.

The cycle cutset method consists of enumerating all the possible instantiations of $\mathcal{C}$, and for each one of them solving the remaining singly connected network by a linear time and space tree algorithm. The instantiations of $\mathcal{C}$ are enumerated by regular OR search, yielding linear space complexity and $O(\exp|\mathcal{C}|)$ time complexity, therefore requiring a minimal cycle cutset to optimize complexity.

A first simple improvement to the traditional cycle cutset scheme described above would be the enumeration of $\mathcal{C}$ by AND/OR search.

Example 3 Figure 3a shows two 3×3 grids, connected on the side node A . A cycle cutset must include at least two nodes from each grid, so the minimal cycle cutset contains three nodes: the common node A and one more node from each grid, for example B and C . The traditional way of solving the cycle cutset problem consists of enumerating all the assignments of the cycle cutset $\{A, B, C\}$, as if these variables form the chain pseudo tree in Figure 3b. However, if A is the first conditioning variable, the remaining subproblem is split into two independent portions, so the cycle cutset $\{A, B, C\}$ can be organized as an AND/OR search space based on the pseudo tree in Figure 3c. If k is the maximum domain size of variables, the complexity of solving Figure 3b is $O(k^3)$ while that of solving Figure 3c is $O(k^2)$.

We can improve the general cycle cutset method, based on the previous example: first find the minimal cycle cutset $\mathcal{C}$; then find the minimal depth start pseudo tree made of nodes in $\mathcal{C}$:

DEFINITION 3 (start pseudo tree) Given an undirected graph $G = (V, E)$, a directed rooted tree $T = (V', E')$, where $V' \subseteq V$, is called a start pseudo tree if it has the same root and is a subgraph of some pseudo tree of G .

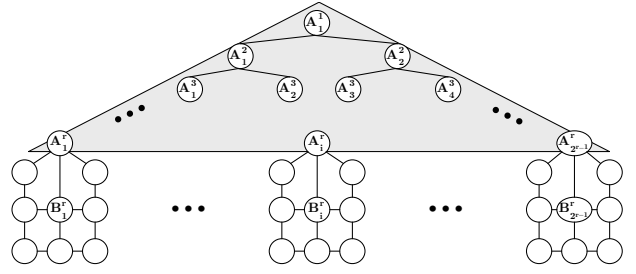


Figure 4: AND/OR cycle cutset

If a cycle cutset of cardinality $|\mathcal{C}| = c$ is explored by AND/OR search, based on a start pseudo tree T over the set $\mathcal{C}$, and the depth of T is m , then $m \leq c$. Therefore,

Proposition 1 Exploring a cycle cutset by AND/OR search is always better than, or the same as, exploring it by OR search.

5 AND/OR Cycle Cutset

The idea presented in section 4 is a straightforward application of the AND/OR paradigm to cycle cutsets. In the following we will describe a more powerful version of the AND/OR cycle cutset.

DEFINITION 4 (AND/OR cycle cutset) Given a graphical model $\mathcal{R} = (X, D, F)$, an AND/OR cycle cutset $AO\text{-}\mathcal{C}$ is a cycle cutset together with an associated start pseudo tree $T_{AO\text{-}\mathcal{C}}$ of depth m . An optimal AND/OR cycle cutset is one having the minimum depth m .

Example 4 Figure 4 shows a network for which the optimal cycle cutset contains fewer nodes than the optimal AND/OR cycle cutset, yet the latter yields an exponential improvement in time complexity. The network in the example is based on a complete binary tree of depth r , the nodes marked A_i^j shown on a gray background. The upper index j corresponds to the depth of the node in the binary tree, and the lower index i to the position in the level. Each of the leaf nodes, from A_1^r to $A_{2^{r-1}}^r$ is a side node in a 3×3 grid. A cycle cutset has to contain at least 2 nodes from each of the 2^{r-1} grids. An optimal cycle cutset is $\mathcal{C} = \{A_1^r, \dots, A_{2^{r-1}}^r, B_1^r, \dots, B_{2^{r-1}}^r\}$, containing 2^r nodes, so the complexity is $O(\exp|\mathcal{C}|) = O(\exp(2^r))$. We should note that the best organization of $\mathcal{C}$ as an AND/OR space would yield a pseudo tree of depth $2^{r-1} + 1$. This is because all the nodes in $\{A_1^r, \dots, A_{2^{r-1}}^r\}$ are connected by the binary tree, so they all must appear along the same path in the pseudo tree (this observation also holds for any other optimal cycle cutset in this example). Exploring $\mathcal{C}$ by AND/OR search lowers the complexity from $O(\exp(2^r))$ to $O(\exp(2^{r-1} + 1))$.

Let's now look at the AND/OR cycle cutset $AO\text{-}\mathcal{C} = \{A_i^j \mid j = 1, \dots, r; i = 1, \dots, 2^{j-1}\} \cup \{B_1^r, \dots, B_{2^{r-1}}^r\}$, containing all the A and B nodes. A pseudo tree in this case is formed by the binary tree of A nodes, and the B nodes exactly in the same position as in the figure. The depth in this case is $r + 1$, so the complexity is $O(\exp(r + 1))$, even though the number of nodes is $|AO\text{-}\mathcal{C}| = |\mathcal{C}| + 2^{r-1} - 1$.

The previous example highlights the conceptual difference between the *cycle cutset method* and what we will call the *AND/OR cycle cutset method*. In *cycle cutset*, the objective is to identify the smallest cardinality cutset. Subsequently, the exploration can be improved from OR search to AND/OR search. In *AND/OR cycle cutset* the objective is to find a cutset that forms a start pseudo tree of smallest depth.

THEOREM 3 *Given a graphical model $\mathcal{R}$, an optimal cycle cutset $\mathcal{C}$, its corresponding smallest depth start pseudo tree $T_{\mathcal{C}}$, and the optimal AND/OR cycle cutset $AO\text{-}\mathcal{C}$ with the start pseudo tree $T_{AO\text{-}\mathcal{C}}$, then:*

$$|\mathcal{C}| \geq \text{depth}(T_{\mathcal{C}}) \geq \text{depth}(T_{AO\text{-}\mathcal{C}}) \quad (1)$$

There exist instances for which the inequalities are strict.

We should note that strict inequalities in Eq. 1 could translate into exponential differences in time complexities.

6 AND/OR w -Cutset

The principle of cutset conditioning can be generalized using the notion of w -cutset. A w -cutset of a graph is a set of nodes such that, when removed, the remaining graph has induced width at most w . A hybrid algorithmic scheme combining conditioning and w -bounded inference was presented in [Rish and Dechter, 2000; Larrosa and Dechter, 2002]. More recently, w -cutset sampling was investigated in [Bidyuk and Dechter, 2003], and the complexity of finding the minimal w -cutset was discussed in [Bidyuk and Dechter, 2004].

The hybrid w -cutset algorithm performs search on the cutset variables and exact inference (e.g. bucket elimination [Dechter, 1999]) on each of the conditioned subproblems. If the w -cutset C_w is explored by linear space OR search, the time complexity is $O(\exp(|C_w| + w))$, and the space complexity is $O(\exp w)$.

The AND/OR cycle cutset idea can be extended naturally to AND/OR w -cutset. To show an example of the difference between the traditional w -cutset and the AND/OR w -cutset we refer again to the example in Figure 4. Consider each 3×3 grid replaced by a network which has a minimal w -cutset C_w . The minimal w -cutset of the whole graph contains in this case $2^{r-1} \cdot |C_w|$ nodes. If this w -cutset is explored by OR search, it yields a time complexity exponential in $(2^{r-1} \cdot |C_w| + w)$. If the w -cutset is explored by AND/OR search it yields a time complexity exponential in $(2^{r-1} + |C_w| + w)$ (similar to Example 4). In contrast to this, the AND/OR w -cutset, which contains the A nodes and the w -cutsets of each leaf network, yields a time complexity exponential only in $(r + |C_w| + w)$, or possibly even less if the nodes in C_w can be organized in a start pseudo tree which is not a chain (i.e. has depth smaller than $|C_w|$).

7 Algorithm Description

The idea of w -cutset schemes is to define an algorithm that can run in space $O(\exp w)$. The AND/OR w -cutset algorithm is a hybrid scheme. The cutset portion, which is organized in a start pseudo tree, is explored by AND/OR search. The remaining w -bounded subproblems can be solved either by

a variable elimination type algorithm, or by search with w -bounded caching - in particular, AND/OR search with full caching is feasible for these subproblems.

7.1 Improved AND/OR Caching Scheme

In [Dechter and Mateescu, 2004], the caching scheme of AND/OR search is based on *contexts* [Darwiche, 2001], which are precomputed based on the pseudo tree before search begins. Algorithm $AO(i)$ performs caching only at the variables for which the context size is smaller than or equal to i (called i -bound).

The cutset principle inspires a more refined caching scheme for AO , which caches some values even at nodes with contexts greater than the i -bound. Let's assume the context of the node X_k is $\text{context}(X_k) = \{X_1, \dots, X_k\}$, where $k > i$. During the search, when variables $X_1, \dots, X_{k-i}$ are instantiated, they can be regarded as part of a cutset. The problem rooted by X_{k-i+1} can be solved in isolation, like a subproblem in the cutset scheme, after the variables $X_1, \dots, X_{k-i}$ are assigned their current values in all the functions. In this subproblem, $\text{context}(X_k) = \{X_{k-i+1}, \dots, X_k\}$, so it can be cached within i -bounded space. However, when the search retracts to X_{k-i} or above, the cache table for variable X_k needs to be purged, and will be used again when a new subproblem rooted at X_{k-i+1} is solved.

This improved caching scheme only increases the space requirements linearly, compared to $AO(i)$, but the time savings can be exponential. We will show results in section 8.

7.2 Algorithm $AO\text{-}C(i)$

We can now define the different versions of AND/OR i -cutset algorithm that we experimented with. We chose to explore the cutset portion either by linear space AND/OR search (no caching) or by AND/OR search with improved caching. For the i -bounded subproblems, we chose either Bucket Elimination (BE) or AND/OR search with full caching (which coincides with the improved caching on the bounded subproblems). The four resulting algorithms are: 1) $AO\text{-}LC(i)$ - linear space cutset and full caching for subproblems; 2) $AO\text{-}LC\text{-}BE(i)$ - linear space cutset and BE for subproblems; 3) $AO\text{-}C(i)$ - improved caching everywhere; 4) $AO\text{-}C\text{-}BE(i)$ - improved caching on cutset and BE on subproblems.

7.3 Finding a Start Pseudo Tree

The performance of $AO\text{-}C(i)$ is influenced by the quality of the start pseudo tree. Finding the minimal depth start pseudo tree for the given i -bound is a hard problem, and it is beyond the scope of this paper to address its complexity and solution. We will only describe the heuristic we used in creating the pseudo trees for our experiments.

Min-Fill [Kjærulff, 1990] is one of the best and most widely used heuristics for creating small induced width orderings. The ordering defines a unique pseudo tree. The minimal start pseudo for an i -bound contains the nodes for which some descendant has adjusted context (i.e. context without the variables instantiated on the current path) greater than i . Min-Fill heuristic tends to minimize context size, rather than

CPCS 422 - $f(i)$											
i	1	2	3	4	5	6	7	8	9	10	11
$d(AO-C)$	32	32	32	32	31	31	31	31	31	30	29
$d(C)$	40	37	32	32	38	37	36	34	32	30	29
$ C $	79	71	65	59	54	50	46	41	37	34	32
GWCA	79	67	60	55	50	46	42	38	34	31	29

Table 1: CPCS 422 - Cutsets Comparison

pseudo tree depth. Nevertheless, we chose to try it and discovered that it provides one of the best pseudo trees for higher values of i .

Min-Depth We developed a heuristic to produce a balanced start pseudo tree, resulting in smaller depth. We start from a Min-Fill tree decomposition and then iteratively search for the separator that would break the tree in parts that are as balanced as possible, relative to the following measure: on either side of the separator eliminate the separator variables, count the number of remaining clusters, say n , and then add the sizes of the largest $\log n$ clusters.

GWC [Bidyuk and Dechter, 2004] is a greedy algorithm to build a minimal cardinality cutset. In the process, we also arranged the minimal cardinality cutset as AND/OR cutset, to compare with the minimal depth cutset that we could find.

8 Experimental Evaluation

We investigated two directions. One was to empirically test the quality of the start pseudo trees, and the other was to compare actual runs of the different versions of $AO-C(i)$.

8.1 The Quality of Start Pseudo Trees

We report here the results on the CPCS 422b network from the UAI repository. It has 422 nodes and induced width 22. Table 1 shows the values of $f(i)$, which expresses the total complexity of a cutset scheme. For a cardinality cutset, $f(i) = i + |C|$ and for an AND/OR cutset of depth d , $f(i) = i + d$. The row $d(AO-C)$ shows the depth of the best AND/OR cutset we could find. $|C|$ shows the number of nodes in the best cutset found by GWC, and $d(C)$ shows its depth when organized as AND/OR cutset. GWCA is taken from [Bidyuk and Dechter, 2004]. The best complexity, expressed by small values of $f(i)$, is always given by the AND/OR cutset, and for smaller values of i they translate into impressive savings over the cardinality cutset C .

In all our experiments described in the following, we refrained from comparing the new cutset scheme with the old cardinality cutset scheme (equivalent to an OR search on the cutset), because the latter was too slow.

8.2 Performance of $AO-C(i)$

We tested the different version of the $AO-C(i)$ family primarily on Bayesian networks with strictly positive distributions, for the task of belief updating. This is necessary to grasp the power of the scheme when no pruning is involved in search.

In all the tables N is the number of nodes, K is the maximum domain size, P is the number of parents of a variable, w^* is the induced width, i is the i -bound, d is the depth of the i -cutset. For most problems, we tested a min-fill pseudo-tree (MF) and one based on the depth minimizing heuristic (MD).

N=40, K=3, P=2, 20 instances, $w^*=7$							
i	Algorithms	d		Time(sec)		# nodes	
		MF	MD	MF	MD	MD	MF
1	AO(i)	12	9	610.14	27.12	50,171,141	1,950,539
	AO-LC(i)			174.53	8.75	13,335,595	575,936
	AO-C(i)			67.99	7.61	4,789,569	499,391
	AO-C-BE(i)			16.95	2.18	-	-
3	AO(i)	7	6	71.68	8.13	5,707,323	595,484
	AO-LC(i)			5.73	0.84	501,793	69,357
	AO-C(i)			2.94	0.84	248,652	69,357
	AO-C-BE(i)			0.69	0.25	-	-
5	AO(i)	4	3	11.28	2.77	999,441	24,396
	AO-LC(i)			0.55	0.54	50,024	4,670
	AO-C(i)			0.55	0.55	49,991	4,670
	AO-C-BE(i)			0.10	0.04	-	-

N=60, K=3, P=2, 20 instances, $w^*=11$							
i	Algorithms	d		Time(sec)		# nodes	
		MF	MD	MF	MD	MD	MF
6	AO-LC(i)	7	6	159.79	63.01	14,076,416	5,165,486
	AO-C(i)			112.43	62.98	9,925,855	5,165,486
	AO-C-BE(i)			27.33	5.50	-	-
	AO-LC(i)			24.40	41.45	2,140,791	3,509,709
9	AO-LC(i)	3	3	24.40	41.45	2,140,791	3,509,709
	AO-C(i)			24.15	40.93	2,140,791	3,509,709
	AO-C-BE(i)			4.27	2.89	-	-
	AO-LC(i)			17.39	38.46	1,562,111	3,173,129
11	AO-LC(i)	0	1	17.66	38.22	1,562,111	3,173,129
	AO-C(i)			17.66	38.22	1,562,111	3,173,129
	AO-C-BE(i)			1.29	2.81	-	-
	AO-C-BE(i)			1.29	2.81	-	-

Table 2: Random Networks

The time and the number of nodes expanded in the search are shown for the two pseudo trees correspondingly.

Random networks. Table 2 shows results for random networks, generated based on N , K and P and averaged over 20 instances. Note that $K=3$, which makes the problems harder, even though w^* seems small. For $N=40$ we see that the old scheme $AO(i)$ is always outperformed. Using improved caching on the cutset is almost always beneficial. For i very close to w^* , caching on the cutset doesn't save much, and in some cases when no caching is possible, the extra overhead may actually make it slightly slower. Also, for strictly positive distributions, switching to BE is faster than running AO search with caching on the remaining problems.

CPCS networks. CPCS are real life networks for medical diagnoses, which are hard for belief updating. Table 3 shows results for CPCS 360 file, having induced width 20. For $i = 20$, $AO-C-BE(i)$ is actually BE. It is interesting to note that $AO-LC-BE(i)$, for $i = 12$ is actually faster than BE on the whole problem, while requiring much less space ($\exp(12)$ compared to $\exp(20)$), due to smaller overhead in caching (smaller cache tables) and a good ordering that doesn't require recomputing the same problems again. We also mention that $AO(i)$ was much slower on this problem and therefore not included in the table.

In the above experiments, the values of d show that MF heuristic provided a better cutset for large values of i , while the MD heuristic provided good cutsets when i was small.

Genetic linkage network. We include in Table 4 results for the genetic linkage network EA4 [Fishelson and Geiger, 2002]. This is a large network, with $N=1173$, but relatively small induced width, $w^* = 15$. This network contains a lot of determinism (zero probability tuples). We did not use in AO search any form of constraint propagation, limiting the algorithm to prune only the zero value nodes (their subproblems do not contribute to the updated belief). We note here that for i -bound 13 and 9, $AO-C(i)$ is faster than $AO-C-BE(i)$ be-

CPCS 360b, N=360, K=2, w* = 20				
i	Algorithms	d (MF)	Time	# nodes
1	AO-LC(i)	23	2,507.6	406,322,117
	AO-LC-BE(i)		1,756.4	-
	AO-C(i)		1,495.2	243,268,549
	AO-C-BE(i)		1,019.4	-
12	AO-LC(i)	8	186.8	14,209,057
	AO-LC-BE(i)		10.3	-
	AO-C(i)		185.1	14,209,057
	AO-C-BE(i)		10.4	-
20	AO-LC(i)	0	167.8	12,046,369
	AO-LC-BE(i)		11.5	-
	AO-C(i)		170.9	12,046,369
	AO-C-BE(i)		11.6	-

Table 3: CPCS 360

EA4 - N=1173, K=5, w*=15							
i	Algorithms	d		Time(sec)		# nodes	
		MF	MD	MF	MD	MD	MF
6	AO(i)	23	21	10.0	103.4	1,855,490	15,312,582
	AO-LC(i)			22.5	76.4	3,157,012	9,928,754
	AO-C(i)			2.0	51.3	281,896	6,666,210
	AO-C-BE(i)			8.4	82.3	-	-
9	AO(i)	18	17	3.3	9.3	410,934	1,466,338
	AO-LC(i)			1.6	4.7	196,662	617,138
	AO-C(i)			1.5	4.8	196,662	616,802
	AO-C-BE(i)			3.5	7.0	-	-
13	AO(i)	3	8	2.0	5.9	235,062	887,138
	AO-LC(i)			1.4	3.6	172,854	431,458
	AO-C(i)			1.6	3.4	172,854	431,458
	AO-C-BE(i)			0.7	5.3	-	-

Table 4: Genetic Linkage Network

cause it is able to prune the search space. We used a version of BE which is insensitive to determinism.

Large networks. Memory limitations are the main drawback of BE. In Table 5 we show results for hard networks, solved by $AO-C-BE(i)$, where $i = 12$ is set to the maximum value that we could use on a 2.4 GHz Pentium IV with 1 GB of RAM. For $N=100$, the space requirements of BE would be about 100 times bigger than the RAM (note $K=3$), yet $AO-C-BE(12)$ could solve it in about six and a half hours, showing the scalability of the AND/OR cutset scheme.

9 Conclusion

The paper presents the *AND/OR w-cutset scheme*, which combines the newly developed AND/OR search for graphical models [Dechter and Mateescu, 2004] with the w-cutset scheme [Bidyuk and Dechter, 2003]. Theorem 3 shows that the new scheme is always at least as good as the existing cutset schemes, but it often provides exponential improvements.

The new AND/OR cutset inspired an improved caching scheme for the AND/OR search, which is always better than the one used by AO(i) [Dechter and Mateescu, 2004], based on context.

The experimental evaluation showed, first, that the theoretical expectations of getting exponential improvements over the traditional cardinality cutset are actually met in practice.

Second, it showed the power and flexibility of the new hybrid scheme. Our conclusion is that improved caching on the cutset is in most cases beneficial. For the remaining problems, if the task is belief updating (or counting solutions) and there is little determinism, then switching to BE is faster. In the presence of determinism, solving the remaining problems with search with full caching may be better. We leave for fu-

K=3, P=2; AO-C-BE(i), i=12			
N	w*	d (MF)	Time(sec)
70	13	2	12
80	15	3	61
90	17	6	2,072
100	18	9	22,529

Table 5: Networks with high memory requirements for BE

ture work the investigation of using look-ahead and no-good learning in the presence of determinism for the AND/OR w-cutset scheme.

Finally, the new scheme is scalable to memory intensive problems, where inference type algorithms are infeasible.

Acknowledgments

This work was supported in part by the NSF grant IIS-0412854 and the MURI ONR award N00014-00-1-0617.

References

- [Bayardo and Miranker, 1996] R. Bayardo and D. Miranker. A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In *AAAI'96*, 1996.
- [Bidyuk and Dechter, 2003] B. Bidyuk and R. Dechter. Cycle-cutset sampling for bayesian networks. In *The Canadian AI Conference, (CAAI'03)*, 2003.
- [Bidyuk and Dechter, 2004] B. Bidyuk and R. Dechter. On finding minimal w-cutset problem. In *UAI'04*, 2004.
- [Darwiche, 2001] A. Darwiche. Recursive conditioning. *Artificial Intelligence Journal*, 125(1-2):5–41, 2001.
- [Dechter and Mateescu, 2004] R. Dechter and R. Mateescu. Mixtures of deterministic-probabilistic networks and their AND/OR search space. In *UAI'04*, 2004.
- [Dechter, 1990] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
- [Dechter, 1999] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, pages 41–85, 1999.
- [Fishelson and Geiger, 2002] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 18(1):189–198, 2002.
- [Freuder and Quinn, 1985] E. C. Freuder and M. J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *IJCAI'85*, 1985.
- [Kjærulff, 1990] U. Kjærulff. Triangulation of graph-based algorithms giving small total state space. Technical report, University of Aalborg, Denmark, 1990.
- [Larrosa and Dechter, 2002] J. Larrosa and R. Dechter. Boosting search with variable-elimination. *Constraints*, pages 407–419, 2002.
- [Pearl, 1988] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.
- [Rish and Dechter, 2000] I. Rish and R. Dechter. Resolution vs. search; two strategies for sat. *Journal of Automated Reasoning*, 24(1/2):225–275, 2000.

Reducing Checks and Revisions in Coarse-grained MAC Algorithms*

D. Mehta[†] and M.R.C. van Dongen
Cork Constraint Computation Centre, Ireland

Abstract

Arc consistency algorithms are widely used to prune the search space of Constraint Satisfaction Problems (CSPs). Coarse-grained arc consistency algorithms like AC-3, AC-3_d and AC-2001 are efficient when it comes to transforming a CSP to its arc-consistent equivalent. These algorithms repeatedly carry out *revisions*. Revisions require *support checks* for identifying and deleting all unsupported values from the domain of a variable. In revisions for difficult problems most values have *some* support. Indeed, most revisions are *ineffective*, i.e. they cannot delete any value and consume a lot of checks and time. We propose two solutions to overcome these problems. First we introduce the notion of a *Support Condition* (SC) which guarantees that a value has *some* support. SCs reduce support checks while maintaining arc consistency during search. Second we introduce the notion of a *Revision Condition* (RC) which guarantees that *all* values have support. A RC avoids a candidate revision and queue maintenance overhead. For random problems, SCs reduce the checks required by MAC-3 (MAC-2001) up to 90% (72%). RCs avoid at least 50% of the total revisions. Combining the two results in reducing 50% of the solution time.

1 Introduction

Arc consistency (AC) algorithms are widely used to prune the search space of binary Constraint Satisfaction Problems (CSPs). MAC [Sabin and Freuder, 1994] is a backtrack algorithm that Maintains Arc Consistency during search. It reduces the thrashing behaviour of a backtrack algorithm, which usually fails many times as a result of the same local inconsistencies. MAC- x uses AC- x for maintaining arc consistency during search.

Many arc consistency algorithms have been proposed. On the one hand there are algorithms such as AC-3 [Mackworth,

1977] and AC-3_d [van Dongen, 2004]. These algorithms have a low $\mathcal{O}(e + n d)$ space complexity but they repeat their support checks and have a non optimal bound of $\mathcal{O}(e d^3)$ for their worst-case time complexity. Here e is the number of constraints, n the number of variables and d the maximum domain size of the variables. On the other hand there are algorithms such as AC-4 [Mohr and Henderson, 1986], AC-6 [Bessière and Cordier, 1993], and AC-2001 [Bessière and Régin, 2001] that use auxiliary data structures to avoid repeating their support checks and have an optimal worst-case time complexity of $\mathcal{O}(e d^2)$.

Since the introduction of AC-4 [Mohr and Henderson, 1986] much work has been done to avoid repeating support checks by using auxiliary data structures. Depending upon the algorithm, these auxiliary data structures store one or more supports for each value involved in each constraint. The belief is that reducing checks helps in solving problems more quickly. However, allowing algorithms to repeat (not too many) checks, relieves them from the burden of a large additional bookkeeping and this may save time, especially if checks are cheap. In this paper, we introduce the notion of a *Support Condition* (SC) which guarantees that a value has *some* support. SCs help avoiding many (but not all) sequences of support checks eventually leading to a support *without storing and maintaining support values*.

Coarse-grained arc consistency algorithms repeatedly carry out revisions to remove all unsupported values from the domain of a variable. Many revisions are *ineffective*, i.e. they cannot remove any value. For example, when RLFP #11 is solved using MAC-3 or MAC-2001, 83% of the total revisions are ineffective. We introduce the notion of a *Revision Condition* (RC) which guarantees that *all* values have some support. RCs help avoiding many (but not all) ineffective revisions and much queue maintenance, which is also a great source of time consumption [van Dongen and Mehta, 2004]. Furthermore, we show that reverse variable based heuristics [Mehta and van Dongen, 2004] result in fewer revisions.

The remainder of this paper is as follows. Section 2 is a brief introduction to constraints. Section 3 discusses revision ordering heuristics and the coarse-grained algorithm AC-3. Sections 4 and 5 introduce the notions of a support condition and a revision condition. Section 6 presents experimental results. Conclusions are presented in Section 7.

*This work has received some support from Science Foundation Ireland under Grant No. 00/PI.1/C075.

[†]The first author is supported by Boole Centre for Research in Informatics.

2 Constraint Satisfaction

A *Constraint Satisfaction Problem* is defined as a set $\mathcal{V}$ of n variables, a non-empty domain $D(x)$ for each variable $x \in \mathcal{V}$ and a set of e constraints among subsets of variables of $\mathcal{V}$. A binary constraint C_{xy} between variables x and y is a subset of the Cartesian product of $D(x)$ and $D(y)$ that specifies the allowed pairs of values for x and y . We only consider CSPs whose constraints are binary.

A value $b \in D(y)$ is called a *support* for $a \in D(x)$ if $(a, b) \in C_{xy}$. Similarly $a \in D(x)$ is called a support for $b \in D(y)$ if $(a, b) \in C_{xy}$. A *support check* (consistency check) is a test to find if two values support each other. A value $a \in D(x)$ is called *viable* if for every variable y constraining x the value a is supported by some value in $D(y)$. A CSP is called *arc-consistent* if for every variable $x \in \mathcal{V}$, each value $a \in D(x)$ is viable.

The *density* p_1 of a CSP is defined as $2e/(n^2 - n)$. The *tightness* p_2 of the constraint C_{xy} between the variables x and y is defined as $1 - |C_{xy}|/|D(x) \times D(y)|$. The *degree* of a variable is the number of constraints involving that variable. Before starting search MAC transforms the input CSP to its arc-consistent equivalent. We call the domain of a variable in this arc-consistent equivalent as the *arc-consistent* domain of that variable. For the remainder of this paper for any variable x , we use $D_{ac}(x)$ for the arc-consistent domain of x , and $D(x)$ for the current domain of x . The *directed constraint graph* of a given CSP is a directed graph having an arc (x, y) for each combination of two mutually constraining variables x and y . We will use G to denote the directed constraint graph of the input CSP.

3 Revision Ordering Heuristics

Coarse-grained arc consistency algorithms use *revision ordering heuristics* to select an arc from a data structure called a queue (set really). When an arc, (x, y) , is selected from the queue, $D(x)$ is *revised* against $D(y)$. Here to *revise* $D(x)$ against $D(y)$ means removing all values from $D(x)$ that are not supported by any value of $D(y)$. Revision ordering heuristics can influence the efficiency of arc consistency algorithms [Wallace and Freuder, 1992]. They can be classified into three categories: *arc based*, *variable based* [McGregor, 1979], and *reverse variable based* [Mehta and van Dongen, 2004] heuristics. The differences between them are as follows.

Arc based revision ordering heuristics are the most commonly presented. Given some criterion they select an arc (x, y) for the next revision. Selecting the best arc can be expensive [van Dongen and Mehta, 2004]. Each selected arc corresponds to exactly one revision. Since there may be many revisions there may be many selections and this may result in a significant overhead. When arc based heuristics are used, the queue needs to be updated after every effective revision unless the domain of a variable becomes empty. Many updates can be an overhead too.

Variable based heuristics [McGregor, 1979] first select a variable x and then repeatedly select arcs of the form (y, x) for the next revision until no more such arcs exist or some $D(y)$ becomes empty. They may be regarded as *propaga-*

Function AC-3: Boolean;

```

begin
  Q := G
  while Q not empty do begin
    select any x from {x : (x, y) ∈ Q}
    effective_revisions := 0
    for each y such that (x, y) ∈ Q do
      remove (x, y) from Q
      revise(x, y, change_x)
      if D(x) = ∅ then
        return False
      else if change_x then
        effective_revisions := effective_revisions + 1
        y'' := y;
    if effective_revisions = 1 then
      Q := Q ∪ {(y', x) | y' is a neighbour of x ∧ y' ≠ y''}
    else if effective_revisions > 1 then
      Q := Q ∪ {(y', x) | y' is a neighbour of x}
  return True;
end;

```

Figure 1: AC-3.

tion based heuristics because the consequences of removing one or more values from $D(x)$ are propagated in all of its neighbours. If a variable x is selected then the domain of all its neighbours in the constraint graph will be revised against $D(x)$. In this setting there are usually fewer selections from the queue at the cost of performing more checks. If checks are cheap then time can be saved because the queue needs fewer selections. Here too every effective revision results in updating the queue.

Reverse variable based heuristics [Mehta and van Dongen, 2004] first select a variable x and then repeatedly select arcs of the form (x, y) for the next revision until there are no more such arcs or $D(x)$ becomes empty. They may be regarded as *support based heuristics* because for one variable x at a time, each value in $D(x)$ seeks support with respect to all of its neighbours for which it is currently unknown whether such support exists. When a variable x is selected a number of revisions is performed which is between 1 and the number of arcs of the form (x, y) currently present in the queue. Therefore, the number of selections (of x), and the overhead of queue management, is usually less.

Selecting a variable x and revising it against all its neighbours y such that (x, y) is currently present in the queue we call a *complete relaxation* of x . Another advantage of using reverse variable based heuristics is that the queue only needs to be updated after every effective *complete relaxation* and not after every effective *revision*, which reduces the number of times the arcs are added to the queue. Overall fewer revisions are performed which results in saving support checks. Pseudo-code for AC-3 equipped with reverse variable based revision ordering heuristic is depicted in Figure 1. The revise function upon which AC-3 depends is depicted in Figure 4.

In Figure 1, if $D(x)$ was changed after a complete relaxation and if this was the result of *only one* effective revision ($effective_revisions = 1$), which happened to be against $D(y'')$, then all arcs of the form (y', x) where y' is a neighbour of x and $y' \neq y''$ are added to the queue. However, if $D(x)$ was changed as the result of *more than one* effective revision ($effective_revisions > 1$) then *all* arcs of the form (y', x) where y' is a neighbour of x are added to the queue. Modulo constraint propagation effects this avoids

queue maintenance overhead.

4 A Support Condition

Arc consistency algorithms are based on the notion of support. Most arc consistency algorithms proposed so far put a lot of effort in *identifying* a support to confirm the existence of a support. Identifying the support is more than is needed to guarantee that a value is supportable: knowing that a support exists is enough. Optimal algorithms like AC-2001 and AC-6 always keep track of the last known support for each value. When this support is lost they try to identify the next support. AC-4 is the only algorithm that confirms the existence of a support by not identifying it during the course of search but its inefficiency lies in its space complexity $\mathcal{O}(e d^2)$ and the necessity of maintaining huge data structures during search.

We propose the notion of a *support condition* (SC), which guarantees that a value has some support. The key point is that it guarantees the existence of a support without identifying it and without storing and maintaining support values. Let C_{xy} be the constraint between x and y , let $a \in D(x)$ (also denoted as (x, a)) and let $\text{scout}[x, y, a]$ be the number of supports of (x, a) in $D_{ac}(y)$. A value $a \in D(x)$ is supported by y if it has at least one support in $D(y)$. Furthermore if $R(y) = D_{ac}(y) \setminus D(y)$ are the removed values from the arc-consistent domain of y then $|R(y)|$ is an upper bound on the number of lost supports of (x, a) in y . Therefore, if the following condition is true then (x, a) is supported by y :

$$\text{scout}[x, y, a] > |R(y)| \quad (1)$$

In order to use the Equation (1) in any coarse-grained MAC algorithm, for each arc-value pair involving the arc (x, y) and the value a on x , $\text{scout}[x, y, a]$ must be assigned the number of supports of (x, a) in $D_{ac}(y)$. Once these support counters are initialised they remain static during search. Hence, there is no overhead of maintaining them. The pseudo-code for computing the support count for each arc-value pair is depicted in Figure 2. In the algorithm, $\text{last}[x, y, a]$ stores AC-2001's last known support for (x, a) in y . Note that the algorithm does not repeat checks and uses the bidirectional property of constraints. For easy problems initialising support counters can be an overhead in terms of support checks. However, it can save time and checks for hard problems.

Next we generalise the idea presented in the Equation (1). Here we associate a weight (non-negative integer) $w[x, y, a]$ with each arc-value pair. If the following condition is true

```

Function InitialiseSupportCounters ( )
  call AC-2001
  if the problem is arc-consistent then
    for each  $(x, y) \in G$  do
      for each  $a \in D_{ac}(x)$  do
         $\text{scout}[x, y, a] := 1$ 
      for each  $(x, y) \in G$  such that  $x < y$  do
        for each  $a \in D_{ac}(x)$  do
          for each  $b \in D_{ac}(y)$  such that  $b > \text{last}[x, y, a]$  do
            if  $b$  supports  $a$  then begin
               $\text{scout}[x, y, a] := \text{scout}[x, y, a] + 1$ 
               $\text{scout}[y, x, b] := \text{scout}[y, x, b] + 1$ 
            end

```

Figure 2: Initialisation of support counters.

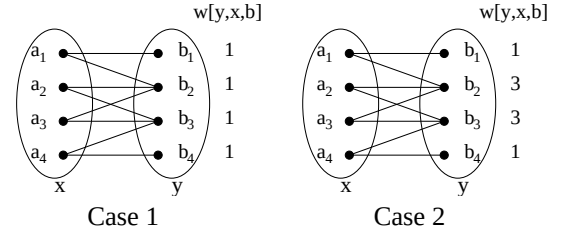


Figure 3: Support inference using different weights

then (x, a) is supported by y :

$$\sum_{(a,b) \in C_{xy}} w[y, x, b] > \sum_{b' \in R(y)} w[y, x, b']. \quad (2)$$

We call the left hand side of Equation (2) the *cumulative weight* of (x, a) with respect to y : it is equal to the sum of the weights of the supports of (x, a) in $D_{ac}(y)$. We call the right hand side of Equation (2) the *removed weight* from $D_{ac}(y)$ with respect to x : it is equal to the sum of the weights of the values removed from $D_{ac}(y)$ with respect to x . Note that if the weight associated with each arc-value pair is 1 then Equation (1) is a special case of the condition in Equation (2). In that case the cumulative weight of (x, a) with respect to y is equal to the number of supports of (x, a) in $D_{ac}(y)$ and the removed weight from $D_{ac}(y)$ is equal to the number of values removed from $D_{ac}(y)$. We call the condition in Equation (2) a *Support Condition* (SC). A SC guarantees the existence of a support and may allow to save many checks.

We illustrate the use of SC to infer the existence of a support by using the example as shown in Figure 3. In Figure 3 (Case 1), the weight associated with each value is 1. The cumulative weight of a_1 with respect to y is 2: it is equal to the number of the supports of a_1 in $D_{ac}(y)$. Now assume that b_1 is deleted from $D_{ac}(y)$ then the removed weight from $D_{ac}(y)$ with respect to x is 1: it equals $|R(y)|$. It can be inferred that a_1 still has a support in $D(y)$ since the cumulative weight of a_1 is greater than the removed weight from $D_{ac}(y)$. Similarly, a support for any value in $D(x)$ can be inferred if any single value is removed from $D_{ac}(y)$. If any two values are removed from $D_{ac}(y)$ then the removed weight from $D_{ac}(y)$ is 2. Here a support cannot be inferred for any value of $D(x)$ because the cumulative weight of each value in $D(x)$ is 2, which is not greater than the removed weight.

If other weights are used and if SC holds for any value $a \in D(x)$ then a support is still guaranteed to exist in $D(y)$. Let us examine Case 2 of Figure 3 where the weight assigned to each value in $D_{ac}(y)$ is its own support count with respect to x . Now the cumulative weight of a_1 is 4 since the weights of the supports of b_1 and b_2 are 1 and 3 respectively. Like Case 1, if any single value is removed from $D_{ac}(y)$, a support can be inferred for all values in $D(x)$. If the two values b_1 and b_4 are removed then the removed weight from $D_{ac}(y)$ is 2. Unlike Case 1 a support can be inferred for all the values of x . In another situation if b_1 and b_2 are removed then also the existence of support can be inferred for at least a_2 and a_3 since their cumulative weights are 6 and the removed weight from $D_{ac}(y)$ is 4. We will see further on that using support

```

Function revise( $x, y$ , var  $change_x$ )
begin
   $change_x := \text{False}$ 
  for each  $a \in D(x)$  do
    if  $\sum_{(a,b) \in C_{xy}} w[y, x, b] > \sum_{b' \in R(y)} w[y, x, b']$  then
      skip /*  $a$  is supported */
    else if  $\nexists b \in D(y)$  such that  $b$  supports  $a$  then begin
       $D(x) := D(x) \setminus \{a\}$ 
       $change_x := \text{True}$ 
    end
end;

```

Figure 4: Algorithm *revise* of AC-3.

counts as weights lets SCs save more checks.

The *revise* function for AC-3 is depicted in Figure 4. This function is slightly different from its original version because it uses SC to avoid a series of checks for which it can be known in advance that it will eventually lead to a support. The use of SC is not restricted to AC-3. It can be integrated in any coarse-grained algorithm.

5 A Revision Condition

The support check is the core operation carried out by arc consistency algorithms. To reduce the number of checks, algorithms proposed so far (1) perform viability checks like in AC-2001, (2) check the status of counters like in AC-4, (3) make some inference based on the supports stored in auxiliary data structures like in AC-7, or (4) carry out SCs as mentioned in Section 4. We call all these tests *auxiliary support checks* (ASCs). When support checks are not too expensive then ASCs can be an overhead and reducing the checks *alone* does not always help in reducing the solution time.

All the improvements proposed so far to reduce support checks are done at fine level of granularity. We propose a *coarser check* at arc level. The idea is that we will use this coarser check to avoid a complete revision. This will not only save support checks but will also avoid *auxiliary* support checks. For a given arc (x, y) , if the least cumulative weight (LCW) of the values of $D(x)$ with respect to y is greater than the removed weight from $D_{ac}(y)$ then all the values of $D(x)$ are supported by y . This can be expressed as follows:

$$\min \left\{ \sum_{(a,b) \in C_{xy}} w[y, x, b] : a \in D(x) \right\} > \sum_{b' \in R(y)} w[y, x, b'] . \quad (3)$$

We call this condition a Revision Condition (RC). The RC can be used by all coarse-grained arc consistency algorithms to reduce unnecessary revisions while maintaining full arc consistency during search.

If a RC holds then it can be exploited *after* selecting the arc (x, y) or when arcs are added to the queue. In the former case the corresponding revision is not carried out and in the latter case the arc (x, y) is not added to the queue. We will use the revision condition by tightening the condition for adding the arcs to the queue: arcs should only be added if RC does not hold. This is depicted in Figure 5. With this implementation the advantages of using the RC are threefold: (a) it will reduce the number of arcs that have to be added to the queue, (b) it will reduce the number of arcs that have to be selected from the queue, and (c) it will reduce the total number of revisions.

```

if  $effective\_revisions = 1$  then
   $Q := Q \cup \{ (y', x) \mid y' \text{ is a neighbour of } x \wedge y' \neq y'' \wedge$ 
     $\min\{ \sum_{(a,b) \in C_{y'x}} w[x, y', b] : a \in D(y') \} \leq \sum_{b' \in R(y')} w[x, y', b'] \}$ 
else if  $effective\_revisions > 1$  then
   $Q := Q \cup \{ (y', x) \mid y' \text{ is a neighbour of } x \wedge$ 
     $\min\{ \sum_{(a,b) \in C_{y'x}} w[x, y', b] : a \in D(y') \} \leq \sum_{b' \in R(y')} w[x, y', b'] \}$ 

```

Figure 5: Enforcing RC while adding the arcs to the queue.

Note that both SC and RC are presented in such a way that the idea is made as clear as possible. This should not be taken as the real implementation. We compute the cumulative weight for each arc-value pair before the search starts. Whenever a value is deleted from the domain of a variable x , we update the removed weight for all the arcs of the form $(\cdot, x)$. If there is a change in the domain size of a variable x after a complete relaxation, we update the LCW for all the arcs of the form $(x, \cdot)$. The space complexity of storing the cumulative weights is $\mathcal{O}(ed)$. The space complexity of storing the removed weights and the LCWs for all the arcs is $\mathcal{O}(e)$ but it may increase to $\mathcal{O}(en)$ during search. The overall space complexity of using SC in conjunction with RC is $\mathcal{O}(ed + en) = \mathcal{O}(e \max(d, n))$.

Updating the least cumulative weight for all the arcs of the form $(x, \cdot)$, if there is a change in the domain of x after every complete relaxation can be an overhead. The other option is to update the LCW of only one arc (x, y) while revising $D(x)$ against $D(y)$ instead of all the arcs of the form $(x, \cdot)$ after a complete relaxation. This can be done cheaply and may avoid the addition of the arc (x, y) to the queue in future. This can be considered as a weak version of a revision condition (WRC) because the LCW is not maintained for all the arcs. The disadvantage here is that not all possible ineffective revisions will be saved then can be saved by RC.

Independent work by [Boussemart *et al.*, 2004] uses $\min\{ \sum_{(a,b) \in C_{xy}} 1 : a \in D_{ac}(x) \} > \sum_{b' \in R(y)} 1$, which is a special case of Equation (3). Note that in their setting $w[x, y, a]$ is 1 for each arc-value pair and the above condition is exploited *after* determining the arc for the revision. For a given arc (x, y) , this condition examines the least cumulative weight (the least support count) of the values of $D_{ac}(x)$ and not of $D(x)$ with respect to y as is done in Equation (3). We call this condition a *static* version of a revision condition because the least cumulative weight is not updated for any arc during the course of search. It remains static for all the arcs. This may not allow to avoid as many ineffective revisions as can be avoided by RC.

6 Experimental Results

6.1 Introduction

In this section, we shall present some results to prove the practical efficiency of SC, RC and WRC. We implemented them in MAC-3 and MAC-2001 equipped with *comp* [van Dongen, 2004] as a revision ordering heuristic. We denote the arc based, reverse variable based, and the variable based heuristic *comp* as *arc:comp*, *rev:comp*, and *var:comp* respectively. The details about these heuristics can be found in

Heuristic	Condition	Weight	MAC-3		Revisions	MAC-2001	
			Checks	Time		Checks	Time
<i>arc:comp</i>	-	-	79,644,545	25.39	16,627,343	20,876,139	27.20
	SC	1	10,226,932	23.22	16,627,343	7,122,238	25.52
	WRC	1	28,632,454	15.51	6,944,423	17,188,921	18.56
	SC + WRC	1	10,226,932	14.49	6,944,423	7,122,238	16.46
<i>rev:comp</i>	-	-	74,985,422	21.76	15,297,543	20,587,469	22.88
	SC	1	10,056,858	19.34	15,297,543	7,064,437	21.23
	SC	<i>scount</i>	7,149,704	26.59	15,297,543	5,569,970	29.14
	WRC	1	28,136,221	13.44	6,708,679	17,063,043	15.80
	WRC	<i>scount</i>	18,548,689	16.21	4,641,513	12,614,384	19.04
	SC + WRC	1	10,056,858	12.21	6,708,679	7,064,437	13.80
	SC + WRC	<i>scount</i>	7,149,704	15.48	4,641,513	5,569,970	17.44
	RC	1	22,256,549	12.74	5,153,349	13,971,627	21.73
	RC	<i>scount</i>	13,902,999	18.03	3,276,247	9,724,569	20.20
	SC + RC	1	10,056,858	12.82	5,153,349	7,064,437	21.45
	SC + RC	<i>scount</i>	7,149,704	18.22	3,276,247	5,569,970	20.27

Table 1: Results for the random problems.

Heuristic	Conditions	MAC-3		Revisions	MAC-2001	
		Checks	Time		Checks	Time
<i>arc:comp</i>	-	56,431,728	2.656	2,154,081	10,412,277	2.661
	SC	36,832,553	2.370	2,154,081	15,217,649	2.626
	WRC	37,250,230	1.771	809,623	14,886,826	1.763
	SC + WRC	36,832,553	1.755	809,623	15,217,649	1.824
<i>rev:comp</i>	-	42,519,506	1.931	1,602,603	10,332,998	2.082
	SC	28,429,473	1.750	1,602,603	15,147,054	2.114
	WRC	28,829,358	1.380	752,277	14,835,844	1.243
	SC + WRC	28,429,473	1.382	752,277	15,147,054	1.270
	RC	28,818,246	1.448	750,789	14,825,485	1.450
	SC + RC	28,429,473	1.467	750,789	15,147,054	1.475

Table 2: Results for RLFAP#11.

[Mehta and van Dongen, 2004]. During search all MACs visited the same nodes in the search tree. They were equipped with a *dom/deg* variable ordering heuristic with a lexicographical tie breaker, where *dom* is the domain size and *deg* is the original degree of a variable. All algorithms were written in C. The experiments were carried out on linux on a PC Pentium III (2.266 GHz processor and 256 MB RAM). Performance is measured in terms of the number of support checks, the CPU time in seconds, and the number of revisions.

We experimented with random problems which were generated by Frost *et al.*'s model B generator [Gent *et al.*, 2001] (<http://www.lirmm.fr/~bessiere/generator.html>). In this model a random CSP instance is characterised by $\langle n, d, p_1, p_2 \rangle$ where n is the number of variables, d the uniform domain size, p_1 the average density, and p_2 the uniform tightness. For each combination of $\langle n, d, p_1, p_2 \rangle$, 50 random problems were generated. Table 1 shows the mean results for $\langle 50, 10, 1.0, 0.13 \rangle$ which is located at the phase transition. In Table 1, under the column labelled as Weight, 1 denotes that the weight associated with each arc-value pair is 1 (also the default value) while *scount* denotes that the weight associated with each arc-value pair is its own support count. Table 2 corresponds to the real world instance RLFAP #11 which came from the CELAR suite. Table 3 corresponds to a quasigroup with holes problem [Achlioptas *et al.*, 2000] of order 10 and 74 holes.

6.2 Discussion

One can immediately notice that SC reduces the number of checks required by MAC-3 and MAC-2001. For instance for the random problem, checks required by MAC-3 and MAC-2001 are reduced by 90% and 72% respectively, when the weight associated with each arc-value pair is its own support count. We observed that for the random problems SC

and RC are especially effective for difficult problems that take a lot of time. The original version of MAC-3 requires 3.81 times more checks than the original version of MAC-2001 but it reduces to 1.28 after using SC. It is interesting to note that for the random problem and the quasigroup problem MAC-3 with SC requires fewer checks than the original version of MAC-2001. Remember that MAC-3 uses a non-optimal algorithm AC-3 while MAC-2001 uses an optimal algorithm AC-2001 but that they usually repeat checks in different branches of the search. In case of RLFAP #11 when SC is used in MAC-2001 there is an improvement in the number of checks required during the course of search but not in the total number of checks. This is caused by the initialisation of the cumulative weights.

MAC-2001 always spends fewer checks than MAC-3 but requires more time. This corresponds to the results presented in [van Dongen, 2004]. Using SC in MAC-3 and MAC-2001 reduces the number of checks but this is not reflected in the solution time. In fact there is only a marginal saving and in some cases it may consume more time. This shows that when checks are cheap carrying out ASCs to reduce the checks is not really a great help in reducing the overall solution time.

Both RC and WRC avoid at least 50% of the total revisions. RCs are able to save more ineffective revisions than WRCs. Satisfying a RC or a WRC avoids a complete revision that not only reduces checks but also ACSs and the overhead of queue management. When *rev:comp* and WRC are used together in MAC-3 or in MAC-2001 there is on average 50% reduction in the solution time compared to the original algorithms equipped with *arc:comp*, which is significant. Results for *arc:comp* with RC are not presented due to the space restriction. The overhead to maintain LCWs is less with *rev:comp* because they need to be updated after every effective complete relaxation but in case of *arc:comp* and *var:comp* after

Heuristic	Conditions	MAC-3		Revisions	MAC-2001	
		Checks	Time		Checks	Time
<i>arc:comp</i>	-	1,760,191,958	202.28	600,538,223	585,239,320	220.88
	SC	444,247,815	204.23	600,538,223	355,925,876	213.84
	WRC	564,731,731	117.20	237,506,329	453,353,800	129.54
	SC + WRC	444,247,815	120.37	237,506,329	355,925,876	129.70
<i>rev:comp</i>	-	1,705,703,347	184.61	573,319,558	571,663,383	203.77
	SC	445,253,641	185.66	573,319,558	356,614,116	198.47
	WRC	565,520,375	110.65	238,239,232	453,868,110	124.16
	SC + WRC	445,253,641	111.33	238,239,232	356,614,116	126.05
	RC	552,491,872	120.65	229,190,153	443,596,880	193.45
	SC + RC	445,253,641	125.97	229,190,153	356,613,391	193.90

Table 3: Results for Quasigroup problems of order 10 with 74 holes.

every effective revision. Results shown in all the tables confirm that *rev:comp* is good in saving revisions, support checks and the CPU time for both MAC-3 and MAC-2001 when compared to *arc:comp*.

Finally, we compared RC with the static version of RC. For the random problems when solved with MAC-3, the static version of RC with *var:comp* as implemented in [Boussemart *et al.*, 2004] spends on average 10,147,799 revisions, 25,328,772 checks and 10.86 seconds while RC with *rev:comp* spends on average 515,349 revisions, 10,056,858 checks and 12.74 seconds. The static version of RC spends about twice as many revisions and checks as RC but saves time. The reason for this is that checks are very cheap for random problems and that the static version uses a variable based queue, which has low maintenance overhead.

7 Conclusions

In this paper first we present a *support condition*. Satisfying SC guarantees the existence of a support for a value. SCs avoid many positive and negative support checks during the course of the search. We showed that when checks are cheap reducing them by using ASCs do not payoff a lot in terms of the CPU time. Instead of carrying out ASCs for each value we tried to reduce the number of ineffective revisions through *revision condition* and a *weak* version of revision condition. Both of them reduce the number of arcs that have to be added to the queue. Having fewer arcs in the queue improves the selection of the best arc from the queue. Furthermore fewer revisions are performed. Overall, SCs reduce the positive and negative support checks required by MAC-3 and MAC-2001. RCs avoid at least 50% of the total revisions. Combining the two results in reducing the solution time by 50%.

Acknowledgements

Thanks to Barbara Smith, Mark Hennessy, Richard Wallace and the other members of the Cork Constraint Computation Centre for helpful comments.

References

[Achlioptas *et al.*, 2000] Dimitri Achlioptas, Carla Gomes, Henry Kautz, and Bart Selman. Generating satisfiable problem instances. In *AAAI/IAAI*, pages 256–261, 2000.

[Bessière and Cordier, 1993] C. Bessière and M. Cordier. Arc-consistency and arc-consistency again. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI’93)*, Washington, DC, 1993.

[Bessière and Régis, 2001] C. Bessière and J.-C. Régis. Refining the basic constraint propagation algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI’2001)*, pages 309–315, 2001.

[Boussemart *et al.*, 2004] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Support inference for generic filtering. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, 2004.

[Gent *et al.*, 2001] I.P. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. *Journal of Constraints*, 6(4):345–372, 2001.

[Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[McGregor, 1979] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.

[Mehta and van Dongen, 2004] D. Mehta and M.R.C. van Dongen. Two new lightweight arc consistency algorithms. In M.R.C. van Dongen, editor, *Proceedings of the First International Workshop on Constraint Propagation and Implementation (CPAI’2004)*, pages 109–123, 2004.

[Mohr and Henderson, 1986] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Sabin and Freuder, 1994] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI’94)*, pages 125–129. John Wiley and Sons, 1994.

[van Dongen and Mehta, 2004] M.R.C. van Dongen and D. Mehta. Queue representation for arc consistency algorithms. In L. McGinty and B. B.Crean, editors, *Proceedings of the Fifteenth Irish Conference on Artificial Intelligence and Cognitive Science*, pages 334–343, 2004.

[van Dongen, 2004] M.R.C. van Dongen. Saving support-checks does not always save time. *Artificial Intelligence Review*, 21(3–4):317–334, 2004.

[Wallace and Freuder, 1992] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of the Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, B.C., 1992.

Applying Local Search to Disjunctive Temporal Problems

Michael D. Moffitt and Martha E. Pollack

Department of Electrical Engineering and Computer Science

University of Michigan

Ann Arbor, MI 48109, USA

{mmoffitt, pollackm}@eecs.umich.edu

Abstract

We present a method for applying local search to overconstrained instances of the Disjunctive Temporal Problem (DTP). Our objective is to generate high quality solutions (i.e., solutions that violate few constraints) in as little time as possible. The technique presented here differs markedly from previous work on DTPs, as it operates within the total assignment space of the underlying CSP rather than the partial assignment space of the related meta-CSP. We provide experimental results demonstrating that the use of local search leads to substantially improved performance over systematic methods.

1 Introduction

Previous work on temporal reasoning has focused primarily on exact and complete methods for efficiently solving temporal problems (e.g., [Stergiou and Koubarakis, 1998; Tsamardinos and Pollack, 2003; Armando *et al.*, 2004]). The purpose of these algorithms is to find a solution that satisfies a set of constraints, or else prove, by means of exhaustive search, that no solution exists. Recently, there has also been work on temporal formalisms where the goal is instead to optimize an objective function; for instance, to maximize a user's preference when the problem permits many solutions [Khatib *et al.*, 2003; Peintner and Pollack, 2004; Morris *et al.*, 2004], or to minimize the number of violated constraints when no complete solution exists [Moffitt and Pollack, 2005]. Although these algorithms are guaranteed to find optimal solutions, they are computationally expensive, and hence may not be applicable to large problems. Given this, it may be advantageous to instead search for approximate solutions when strict optimality is not required.

Local search algorithms are known to be among the most effective methods for solving computationally intractable problems such as propositional satisfiability, scheduling, and constraint satisfaction [Hoos and Stutzle, 2004]. While they do not guarantee optimality, they are often able to produce high-quality solutions within a short amount of time. Despite the significant attention that local search has received in combinatorial optimization, its application to problems of quantitative temporal reasoning has as yet been largely overlooked.

In this paper, we show how local search can be successfully applied to overconstrained instances of the Disjunctive Temporal Problem (DTP) [Stergiou and Koubarakis, 1998], a particularly expressive form of temporal constraint satisfaction problem. While local search has been commonly applied to finite-domain CSPs, its application to DTPs is of particular interest, as the domains of the temporal variables are infinite, ranging over the entire set of either integers or real numbers. Previous work on DTP-solving has focused on a reformulation of the problem, in which a meta-CSP is constructed and searched. In contrast, we apply local search directly to the original CSP, i.e., the DTP itself. We discuss issues such as solution initialization, neighbor generation, and cost computation, and provide experimental results demonstrating that the use of local search in solving overconstrained DTPs can lead to substantially improved performance as compared to systematic methods.

It should be noted that ours is not the first attempt to apply local search to temporal reasoning. This work is closely related to [Beaumont *et al.*, 2004]; however, that line of research deals with a qualitative interval algebra representation, which is strictly less expressive than that allowed by DTPs. Nonetheless, their endpoint-ordering technique bears similarity to our approach, in that it also abandons the meta-CSP for the original CSP search space. Our work is also related to [Walser, 1998], where local search is applied to overconstrained integer programs without disjunctions.

2 Disjunctive Temporal Problems

A *Disjunctive Temporal Problem* (DTP) is a constraint satisfaction problem defined by a pair $\langle X, C \rangle$, where each element $X_i \in X$ designates a time point, and C is a set of constraints of the following form:

$$c_{i1} \vee c_{i2} \vee \dots \vee c_{in}$$

where in turn, each c_{ij} is of the form $x - y \leq b$; $x, y \in X$ and $b \in \mathbb{R}$. (In practice, b is often restricted to the integers.) DTPs are thus a generalization of Simple Temporal Problems (STPs), in which each constraint is limited to a single inequality [Dechter *et al.*, 1991]. A solution to a DTP is an assignment of values to time points such that all constraints are satisfied.

Several algorithms have been developed for solving DTPs [Stergiou and Koubarakis, 1998; Armando *et al.*, 1999;

[Oddi and Cesta, 2000; Tsamardinos and Pollack, 2003]. Typically, these algorithms transform the problem into a *meta-CSP*, in which the original DTP is viewed as a collection of alternative STPs. Using this approach, the algorithm selects a single disjunct from each constraint of a given DTP. The resulting set forms an STP, called a *component STP*, which can then be checked for consistency in polynomial time using a shortest-path algorithm. Clearly, a DTP D is consistent if and only if it contains at least one consistent component STP. Furthermore, any solution to a consistent component STP of D is also a solution to D itself. Consequently, it is standard in the DTP literature to consider any consistent component STP to be a solution of the DTP of which it belongs.

A number of pruning techniques can be used to focus the search for a consistent component STP of a DTP, including conflict-directed backjumping, removal of subsumed variables, and semantic branching. The DTP solver Epilitis [Tsamardinos and Pollack, 2003] integrated all these techniques, in addition to no-good recording. At the time it was developed, Epilitis was the fastest existing DTP solver, though it was recently surpassed by TSAT++ [Armando *et al.*, 2004].

DTP solvers such as Epilitis perform total constraint satisfaction — that is, their objective is to find a solution that satisfies all the constraints of a DTP. In the event that a DTP is inconsistent, these solvers are capable of detecting such infeasibility, but are incapable of providing partial solutions that come close to satisfying the problem. In response, the DTP solver Maxilitis [Moffitt and Pollack, 2005] was designed to find solutions that maximize the number of satisfied constraints. Partial constraint satisfaction has the disadvantage of being expensive, as the pruning techniques typically used in DTP solving become weaker when relaxations are allowed. If the overconstrained DTP is very large, the systematic search that Maxilitis performs becomes intractable, and one must instead settle for an approximate solution. Fortunately, Maxilitis is an anytime algorithm, and one can interrupt it at any point to extract a (possibly suboptimal) partial assignment. It is not, however, obvious that a systematic search is the fastest way to find an approximate solution.

3 Local Search

When exact, systematic methods for solving hard combinatorial problems are too expensive, local search offers an alternative approach for quickly generating approximate solutions. Applications of local search are numerous, and include planning [Ambite and Knoblock, 1997], scheduling [Storer *et al.*, 1992], and constraint satisfaction [Minton *et al.*, 1992].

The Satisfiability Problem (SAT) is a classic domain for application of local search. The objective of SAT is to find an assignment to a set of binary variables that satisfies a Boolean formula F , typically given in Conjunctive Normal Form (CNF). A common variation on SAT is MAX-SAT, where the objective is to maximize the number of satisfied clauses in F . A number of local search algorithms have been constructed for both SAT and MAX-SAT. One of the most successful variants is GSAT [Selman *et al.*, 1992]. In this algorithm, one begins with a random assignment of truth values

to the propositional variables, and then repeatedly chooses a variable to flip that results in the maximal decrease in violated clauses, breaking ties randomly. To avoid getting stuck in local minima, a technique called *random restarts* is often used, where the algorithm begins anew with another random assignment. Several variants exist for GSAT; of these, GSAT/Tabu [Mazure *et al.*, 1997] is among those with best performance. It maintains a *tabu list* of recently flipped variables, which are then temporarily prohibited from being selected for another flip.

The basic method used in GSAT and GSAT/Tabu can be applied to other problems besides SAT. Indeed, the structure of a DTP closely resembles that of a SAT problem, and it is for this reason that a SAT solving approach has been applied to the problem of finding complete solutions to DTPs (as in TSAT [Armando *et al.*, 1999] and TSAT++ [Armando *et al.*, 2004]).

3.1 Application to DTPs

In applying local search to possibly overconstrained DTPs, we again need to decide whether to work in the meta-CSP or in the original CSP (the DTP). To illustrate these candidates, consider the following very small problem:

$$\begin{aligned} C_1 &: \{c_{11} : a - b \leq 10\} \\ C_2 &: \{c_{21} : b - a \leq -15\} \vee \{c_{22} : c - a \leq -25\} \\ C_3 &: \{c_{31} : b - c \leq 10\} \\ C_4 &: \{c_{41} : a - c \leq 20\} \end{aligned}$$

Clearly there is no complete solution to this problem, since c_{21} conflicts with c_{11} , and c_{22} conflicts with the constraint induced by the transitive composition of c_{11} and c_{31} . As a result, our objective will be to find a solution that maximizes the number of constraints satisfied in the DTP. We define the cost of a solution to be the number of constraint violations it induces; thus, low-cost solutions are better.¹

Partial Assignment Space of the Meta-CSP

One way to view a solution to overconstrained DTPs such as the example above is as a *partial* assignment to the meta-CSP, in which some of the meta-level variables are left unassigned. For example, consider the assignment $(C_1, C_2, C_3, C_4) \leftarrow (c_{11}, c_{22}, \epsilon, \epsilon)$, in which the constraints C_3 and C_4 are not given an assignment (indicated by ϵ). Here the (partial) component STP that is selected consists of c_{11} and c_{22} , and has a cost of 2. This particular partial assignment cannot be extended to any solution of lower cost, since inclusion of either of the disjuncts c_{31} or c_{41} would result in an inconsistency. However, this does not mean that a solution of higher quality does not exist. Indeed, the partial assignment $(c_{11}, \epsilon, c_{31}, c_{41})$ with cost 1 is both consistent and optimal.

While search in the partial assignment space of the meta-CSP is common to most systematic methods for solving DTPs, it is less attractive for the application of local search. First, systematic methods work within a backtracking tree, where disjuncts are removed in the order in which they were

¹Other cost functions — for instance, the maximum deviation from the bound of any constraint — are possible. However, there currently exist no systematic methods to compare against that minimize such alternative cost functions.

added.² One technique that is commonly used in DTP solving, *incremental full-path consistency* [Mohr and Henderson, 1986], exploits this property by maintaining a stack of the temporal network updates made during search, using it to cheaply repair path dependencies when backtracking. Local search requires the ability to modify arbitrary values in the partial assignment, not necessarily respecting the order in which they were originally assigned; thus it cannot exploit the incremental approach. Second, several of the powerful pruning techniques used by DTP solvers such as Epilitis and Maxilitis have no meaning outside the context of a systematic search tree. For example, semantic branching is able to acquire additional network constraints by exhaustively exploring particular assignments of disjuncts to constraints. In local search, no such exhaustive search is performed, and consequently it is hard to imagine how to adapt this mechanism. Finally, in local search, the number of neighbors for a partial assignment will typically be much larger than the number of successors in systematic search, thus making evaluation of alternatives prohibitively expensive.

Total Assignment Space of the Original CSP

An alternative approach is to perform search in the space of total assignments to the time points in the original CSP. For the example DTP above, one possible assignment is $(a, b, c) \leftarrow (30, 25, 0)$. Since this solution violates constraints C_3 and C_4 it has a cost of 2. A better solution would be $(20, 10, 0)$, which violates only constraint C_2 .

A key issue for this search space is how to define the neighbors of an assignment. One approach is to include all solutions whose variable assignments differ in exactly one position; for example, $(a, b, c) \leftarrow (30, 15, 0)$ would be a neighbor of $(a, b, c) \leftarrow (30, 25, 0)$, since the two differ only on the assignment to b . However, because the value of each variable may take any real (or any integer), the cardinality of the set of possible neighbors is uncountably (or countably) infinite. It is for this reason that most of the prior work on DTPs has avoided search in this space: the infinite cardinality of the variables' domains precludes the use of methods for exhaustive search. We address this concern in the next section, where we describe our local search algorithm in detail.

4 Localitis

In this section, we present Localitis, our algorithm for using local search to perform partial constraint satisfaction in a DTP. Its basic framework derives from the GSAT/Tabu algorithm that is commonly used for SAT and MAX-SAT instances. It also bears considerable resemblance to WalkSAT(OIP), used in [Walser, 1998] to solve overconstrained integer programs without disjunctions³.

²With chronological backtracking, the last disjunct added is the first one removed; with more sophisticated techniques such as conflict-directed backjumping, additional disjuncts may be removed at the same time, but the removal always follows the structure of the backtracking tree.

³While any disjunctive program can be cast as an integer program, the conversion process requires the creation of additional variables and constraints. Our method instead operates directly on the constraints in their disjunctive form.

4.1 Generating the Initial Solution

Typically in algorithms such as GSAT, the initial solution is chosen by selecting random assignments for each of the Boolean variables. A similar initialization can be done with DTPs: each time point can be randomly assigned a value within some interval $[L, U]$. If one employs random restarts, this initialization can be done several times to ensure adequate exploration of the search space.

A potentially better alternative is to make a greedy assignment to the time points, so that the algorithm begins with a reasonable solution. We adopt this approach in Localitis. To generate the initial solution, we make use of the Maxilitis solver. Recall that Maxilitis searches in the meta-CSP space. We invoke it and let it run until it generates its first solution, in which every constraint is assigned either a disjunct or ϵ . Since this solution is not necessarily optimal, Maxilitis would normally continue, but we instead terminate it, and project an assignment to time points from the component STP it has computed, making random assignments to all time points outside the scope of the component STP. Because of the variable and value ordering heuristics used by Maxilitis, the first solution it encounters is likely to be better than what a purely random assignment would provide. One could repeat this process for random restarts, placing bias on the selection of assignments so that the set of disjuncts chosen differs between runs.

4.2 The Neighborhood

As described earlier, the most obvious way to define the neighbor(s) of an assignment is as the set of assignments that differ in the value of only one time point (for example, $(a, b, c) \leftarrow (30, 25, 0)$ and $(a, b, c) \leftarrow (30, 15, 0)$). Unfortunately, this definition results in each assignment having infinitely many neighbors. The key to reducing the size of the search space is to note that if we hold fixed the values of all the variables but one (b in the current example), then only a small set of new values for the selected variable are significant. Specifically, we only need to consider those values for which the *slack* of a disjunct belonging to some constraint becomes zero — that is, when an inequality becomes a strict equality. This is somewhat similar to a pivot step in the simplex method for linear programming, which maintains a basic feasible solution that corresponds to an “active system” of constraints in the LP. For instance, in our running example, suppose we are given the assignment of $(30, 25, 0)$. Then the significant values for b are 20, 15, and 10, as they would make the disjuncts c_{11} , c_{21} , and c_{31} active, respectively. Some of these values may satisfy several new disjuncts at once (for instance, if b is assigned the value 10, it satisfies disjuncts c_{21} and c_{31} simultaneously). No other values for b are capable of satisfying a set of constraints that one of these significant values cannot.

We can impose yet another restriction on the set of neighbors; namely, that they include only those new, significant values that change the resulting set of satisfied constraints. For instance, while the assignment of 20 to b is significant in that it makes the slack for c_{11} zero, it does not change the set of satisfied constraints. As a result, this new assignment is not particularly interesting, and need not be considered in the set of neighbors of this assignment.

4.3 Neighbor Selection and Tabu Moves

Since our objective is to satisfy the maximum number of constraints, selection of which neighbor to explore is a fairly straightforward process: choose the candidate that satisfies the maximum number of constraints possible. This is identical to GSAT, where the variable chosen to flip is the one that will minimize the number of unsatisfied clauses. In the case that several assignments would result in the same minimal number of violations, one is selected at random.

When a local search process is required to continually make greedy moves, it can easily get stuck in local minima. To avoid this, we adopt a common variation on the technique of tabu search [Glover and Laguna, 1997], and forbid local search to change the values of variables that were recently modified. A parameter tt , called the tabu tenure, determines the duration (in search steps) for which this restriction applies. When tabus are applied to SAT, no variable flipped at iteration number i is allowed to flip again until tt steps have passed. We introduce tabu moves into Localitis in a similar way. If the value of a variable x is changed at step i , its value is then fixed until tt steps have passed. To efficiently determine the tabu status of each variable x , we maintain an array it_x , where it_x stores the search step number when variable x was last changed. A variable is tabu if and only if $i - it_x < tt$.

4.4 Efficient Cost Computation

The simplest way to compute the cost of a neighbor assignment is to temporarily enforce the assignment being considered, and subsequently test all constraints for satisfiability, counting the number that are not satisfied. However, this straightforward implementation has been shown to be rather inefficient, since many of the constraints will not be affected by the local modification [Selman *et al.*, 1992].

A common technique is to instead compute the relative change in cost by testing only those clauses that contain the variable in question. This can be facilitated by a preprocessing step which creates a list for each variable, containing indices for those constraints that it participates in.

4.5 The Algorithm

Figure 1 provides the pseudocode for Localitis. The function *Maxilitis-First-Path()* is used to generate the initial assignment (line 1). The algorithm then performs local search, generating the neighbors of the current assignment by looking at the non-tabu variables in each disjunct of the DTP (lines 4-5), and considering “moves”, i.e., assignments to significant values (line 6) that change the set of satisfied constraints (line 7). (The notation X/Y , where Y is an assignment $Z \leftarrow z$, denotes the substitution of Y for the original assignment to Z in X . The function *Sat()* returns the set of constraints in the DTP that are satisfied by its argument.) The set *moves* stores the set of minimal-cost neighbors (lines 8-13), from which one is eventually selected at random (lines 19-20). After storing the value of the best solution seen to date (lines 21-22) and updating tabu tenures (line 24), the process iterates.

5 Experimental Results

We implemented Localitis and conducted a set of experiments whose primary goals were 1) to determine the influence of

Localitis(DTP D)

```

1. Best-Assign  $\leftarrow$  Assign  $\leftarrow$  Maxilitis-First-Path( $D$ )
2. For  $it = 1$  to  $max\_steps$ 
3.    $min\_cost \leftarrow \infty$ ,  $moves \leftarrow \emptyset$ 
4.   For each disjunct  $d: x - y \leq b$ 
5.     If  $x$  not tabu
6.        $move \leftarrow (x \leftarrow y + b)$ 
7.       If  $Sat(Assign) \neq Sat(Assign/move)$ 
8.         If  $cost(Assign/move) = min\_cost$ 
9.            $moves \leftarrow moves \cup \{move\}$ 
10.        EndIf
11.        If  $cost(Assign/move) < min\_cost$ 
12.           $moves \leftarrow \{move\}$ 
13.           $min\_cost \leftarrow cost(Assign/move)$ 
14.        EndIf
15.      EndIf
16.    EndIf
17.    Repeat lines 5 – 16 for  $y$  (i.e.,  $y \leftarrow x - b$ )
18.  EndFor
19.   $new\_move \leftarrow Random-Member(moves)$ 
20.   $Assign \leftarrow Assign/new\_move$ 
21.  If  $cost(Assign) < cost(Best-Assign)$ 
22.     $Best-Assign \leftarrow Assign$ 
23.  EndIf
24.  Update tabu count for the time point in  $new\_move$ 
25. EndFor
26. return Best-Assign

```

Figure 1: Localitis, a local search algorithm for DTPs

the way in which the initial assignment is made, 2) to analyze the effect of various neighborhood functions and tabu tenures, and 3) to compare the anytime quality of this solver against its cousin Maxilitis, which performs an exhaustive search of the DTP in the meta-CSP space. No random restarts were used for these tests. To benchmark our algorithm, we used DTPs created by a random generator used in testing previous DTP solvers [Stergiou and Koubarakis, 1998]. The test case generator takes as arguments the parameters $\langle k, N, m, L \rangle$, where k is the number of disjuncts per constraint, N is the number of time points, m is the number of constraints, and L is the constraint width, i.e., a positive integer such that for each disjunct $x - y \leq b$, $b \in [-L, L]$ with uniform probability. In our experiments, we set $k = 2$, $N = 25$, $m = 175$, and $L = 100$. A derived parameter R (the ratio of constraints over variables, m/N) expressing constraint density was thus 7. For this set of parameters, 50 random problems were generated. The domains of the variables are integers instead of reals, which again is standard in DTP literature. Our implementation of Localitis was developed in Java, and our experiments were conducted on a 3 GHz Intel Pentium 4 machine running Windows XP and having 1 GB of memory.

In our first experiment, we tested 5 different values for the tabu tenure to measure its effect on solution quality as a function of time. In Figure 2, we plot a curve for each tabu tenure in the set $\{0, 1, 2, 3, 4\}$. The number of seconds elapsed is shown on the x -axis, and the number of constraint violations in the solution (averaged over the 50 test cases) is shown on

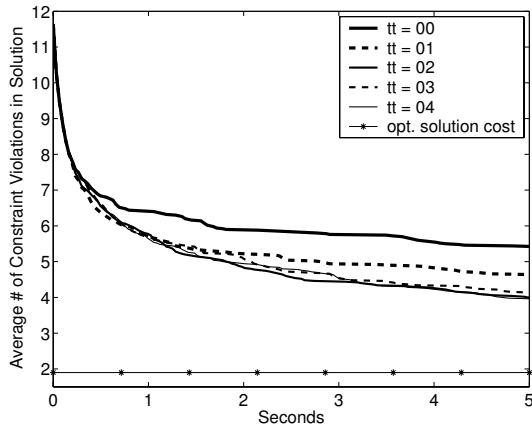


Figure 2: Anytime curves for various tabu tenures

the y -axis. We also show the optimal solution cost, which expresses the average number of violations in the optimal solution (i.e., what Maxilitis reported when run until completion). Convergence is slowest for when the tabu tenure is 0 (this effectively corresponds to the absence of tabu search), and is also somewhat slow for when the tabu tenure is 1. For values 2, 3, and 4, convergence is almost identical. Larger values (not shown) displayed no improvement over these curves, and so a tabu tenure of 2 was used for all subsequent tests.

In our second experiment, we studied two different strategies for generating the initial set of temporal values: one using random assignments, uniformly chosen from the range $[-L, L]$, and the other using Maxilitis to greedily generate an initial solution, as described previously. The results are shown in Figure 3 (among other results that will be addressed momentarily). Once again, the x - and y -axes represent the seconds elapsed and the solution cost, respectively. The curve labeled ‘Localitis (Normal)’ uses the greedy selection, and the curve labeled ‘Localitis (Random Initialization)’ does not. The former begins with an average cost of 11.64 violated constraints, where the latter begins with a drastically higher average of 43.60 not shown on the graph. The randomly initialized search is able to make up the difference fairly quickly, although it continues to lag for the duration of the search. The shape of the curves are roughly identical, despite the horizontal displacement. Thus, it appears that the greedy initialization does indeed improve the starting solution, although the effect on the convergence rate is negligible.

In our third experiment, we tested two different neighborhood criteria: the first includes all assignments of significant values (ones that force the slack of some disjunct to be zero), while the second also requires that there be some change in the set of satisfied constraints in the DTP. The results are shown in Figure 3. The curve labeled ‘Localitis (Normal)’ uses the more restricted definition, and the curve labeled ‘Localitis (Any 0-Slack move)’ does not. While the two begin at the same initial point (as they should, since both use greedy initialization), the ‘Localitis (Normal)’ curve is able to generate solutions of higher quality much earlier. For instance, after 5 seconds, ‘Localitis (Any 0-Slack move)’ has gener-

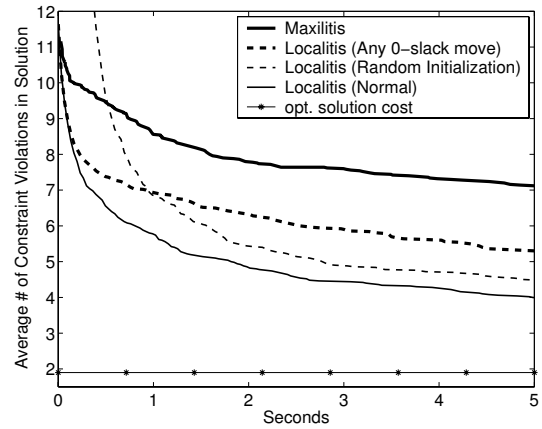


Figure 3: Anytime curves for Maxilitis & Localitis variations

ated an average solution cost of 5.3. Only 1.28 seconds were required for ‘Localitis (Normal)’ to obtain this same average. As a result, it seems that one can expect significantly better results when the neighborhood is restricted to only those candidates which modify the partial assignment of the meta-CSP.

Finally, we compare the anytime quality of our Localitis solver to that of the systematic solver Maxilitis, which is (at present) the only other existing method for performing partial constraint satisfaction of DTPs. The results are again shown in Figure 3. The curves of interest are labeled ‘Localitis (Normal)’ and ‘Maxilitis.’ By inspection, it seems that despite its numerous pruning techniques and sophisticated constraint propagation methods, Maxilitis is far slower at producing high quality solutions. For instance, at the end of the 5 seconds shown, Maxilitis has generated an average solution cost of about 7.12. For Localitis to produce the same average cost requires only 0.29 seconds. To reach the average cost of 4 that Localitis achieves after 5 seconds requires 104 seconds of Maxilitis runtime. Based on these observations, the speedup achieved appears to be roughly 17 to 20 times faster.

6 Discussion and Future Work

In this paper, we have presented a method for applying local search to overconstrained instances of the Disjunctive Temporal Problem. In contrast to previous algorithms for solving DTPs, our technique abandons the meta-CSP and instead explores the total assignment space of the underlying CSP. Our results show that the computation time required to generate high-quality solutions is significantly reduced in comparison to traditional branch-and-bound algorithms for performing partial constraint satisfaction.

Given the freedom allowed by this alternative search space, one particularly interesting avenue of research would be to extend this approach toward more expressive cost functions. Whereas this paper concentrates on minimizing the number of violated constraints, one could instead capture the amount by which the constraints are violated. This would give higher value to those solutions whose assignments come close to falling within the prescribed bounds.

Another appealing possibility is to apply this same local search technique to underconstrained rather than overconstrained temporal formalisms. The recent addition of preferences to DTPs, appropriately labeled DTPs with Preferences (DTPPs) [Peintner and Pollack, 2004], allows preference functions to be defined over particular values of the temporal differences. As no efficient optimal algorithm is yet known for maximizing the weighted sum of preferences in DTPPs, greedy methods are currently being developed to generate approximate solutions. Local search may indeed prove to be a competitive alternative to these algorithms.

Acknowledgments

The authors thank Neil Yorke-Smith, Dushyant Sharma, and Bart Peintner for their input into this work. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010 and the Air Force Office of Scientific Research under Contract No. FA9550-04-1-0043. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of DARPA, the Department of Interior-National Business Center, or the United States Air Force.

References

- [Ambite and Knoblock, 1997] Jose Luis Ambite and Craig A. Knoblock. Planning by rewriting: Efficiently generating high-quality plans. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 706–713, 1997.
- [Armando et al., 1999] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European Conference on Planning*, pages 97–108, 1999.
- [Armando et al., 2004] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, and Marco Maratea. A SAT-based decision procedure for the boolean combination of difference constraints. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-2004)*, 2004.
- [Beaumont et al., 2004] Matthew Beaumont, John Thornton, Abdul Sattar, and Michael Maher. Solving overconstrained temporal reasoning problems using local search. In *Proceedings of the 8th Pacific Rim Conference on Artificial Intelligence (PRICAI-2004)*, 2004.
- [Dechter et al., 1991] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.
- [Glover and Laguna, 1997] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, MA, USA, 1997.
- [Hoos and Stutzle, 2004] Holger Hoos and Thomas Stutzle. *Stochastic Local Search: Foundations and Applications*. Kluwer Academic Publishers, 2004.
- [Khatib et al., 2003] Lina Khatib, Paul Morris, Robert Morris, and K. Brent Venable. Tractable Pareto optimal optimization of temporal preferences. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, pages 1289–1294, 2003.
- [Mazure et al., 1997] Bertrand Mazure, Lakhdar Sais, and Eric Gregoire. TWSAT: A new local search algorithm for SAT - performance and analysis. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 281–285, 1997.
- [Minton et al., 1992] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.
- [Moffitt and Pollack, 2005] Michael D. Moffitt and Martha E. Pollack. Partial constraint satisfaction of disjunctive temporal problems. In *Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference (FLAIRS-2005)*, 2005.
- [Mohr and Henderson, 1986] Roger Mohr and Thomas C. Henderson. Arc-consistency and path-consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
- [Morris et al., 2004] Paul Morris, Robert Morris, Lina Khatib, Sailesh Ramakrishnan, and A. Bachmann. Strategies for global optimization of temporal preferences. In *Proceedings of the 10th International Conference on Principles and Practices of Constraint Programming*, pages 408–422, 2004.
- [Oddi and Cesta, 2000] Angelo Oddi and Amedeo Cesta. Incremental forward checking for the disjunctive temporal problem. In *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 108–112, 2000.
- [Peintner and Pollack, 2004] Bart Peintner and Martha E. Pollack. Low-cost addition of preferences to DTPs and TCSPs. In *Proceedings of the 19th National Conference on Artificial Intelligence*, pages 723–728, 2004.
- [Selman et al., 1992] Bart Selman, Hector J. Levesque, and David G. 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.
- [Stergiou and Koubarakis, 1998] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 248–253, 1998.
- [Storer et al., 1992] Robert H. Storer, S. David Wu, and Renzo Vaccari. New search spaces for sequencing problems with application to job shop scheduling. In *Management Science*, volume 38, pages 1495–1509, 1992.
- [Tsamardinos and Pollack, 2003] Ioannis Tsamardinos and Martha E. Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151(1-2):43–90, 2003.
- [Walser, 1998] Joachim P. Walser. *Domain-independent Local Search for Linear Integer Optimization*. PhD thesis, Universitat des Saarlandes, 1998.

Possibilistic Stable Models

Pascal Nicolas and Laurent Garcia and Igor Stéphan

LERIA, University of Angers, France

{pascal.nicolas, laurent.garcia, igor.stephan}@univ-angers.fr

Abstract

In this work, we define a new framework in order to improve the knowledge representation power of Answer Set Programming paradigm. Our proposal is to use notions from possibility theory to extend the stable model semantics by taking into account a certainty level, expressed in terms of necessity measure, on each rule of a normal logic program. First of all, we introduce possibilistic definite logic programs and show how to compute the conclusions of such programs both in syntactic and semantic ways. The syntactic handling is done by help of a fix-point operator, the semantic part relies on a possibility distribution on all sets of atoms and we show that the two approaches are equivalent. In a second part, we define what is a possibilistic stable model for a normal logic program, with default negation. Again, we define a possibility distribution allowing to determine the stable models.

1 Introduction

Answer set programming (ASP) is an appropriate formalism to represent various problems issued from Artificial Intelligence and arising when available information is incomplete as in non-monotonic reasoning, planning, diagnosis... From a global view, ASP is a general paradigm covering different declarative semantics for different kinds of logic programs. One is the *stable model semantics* [Gelfond and Lifschitz, 1988] for definite logic programs augmented with default negation. In ASP, information is encoded by logical rules and solutions are obtained as sets of models. Each model is a minimal set of atoms representing sure informations (some facts) and deductions obtained by applying by default some rules. So, conclusions rely on present and absent information, they form a coherent set of hypotheses and represent a rational view on the world described by the rules. Thus, in whole generality there is not a unique set of conclusions but maybe many ones and each conclusion is no longer absolutely sure but only plausible and more or less certain.

Possibilistic logic [Dubois *et al.*, 1995] is issued from Zadeh's possibility theory [Zadeh, 1978], which offers a framework for representation of states of partial ignorance owing to the use of a dual pair of possibility and necessity

measures. Possibility theory may be quantitative or qualitative [Dubois and Prade, 1998] according to the range of these measures which may be the real interval $[0, 1]$, or a finite linearly ordered scale as well. Possibilistic logic provides a sound and complete machinery for handling qualitative uncertainty with respect to a semantics expressed by means of possibility distributions which rank-order the possible interpretations. Let us mention that in possibilistic logic we deal with uncertainty by means of classical 2-valued interpretations that can be more or less certain.

The aim of our work is to introduce the concepts of possibilistic logic into ASP in order to deal with uncertainty. This can be illustrated by the following example of a medical treatment in which a patient is suffering from two diseases. Each disease can be cured by one drug but the two drugs are incompatible. The program $P = \{dr1 \leftarrow di1, not\ dr2., dr2 \leftarrow di2, not\ dr1., c1 \leftarrow dr1, di1., c2 \leftarrow dr2, di2., di1., di2.\}$ means that the drug $dr1$ (resp. $dr2$) is given to a patient if he suffers from the disease $di1$ (resp. $di2$) except if he takes the drug $dr2$ (resp. $dr1$); if a patient suffering from disease $di1$ (resp. $di2$) takes the drug $dr1$ (resp. $dr2$) then he is cured $c1$ (resp. $c2$) of his disease; the patient suffers from the diseases $di1$ and $di2$. From this program, two stable models $\{di1, di2, dr1, c1\}$ and $\{di1, di2, dr2, c2\}$ are obtained. However, it seems interesting for a doctor to evaluate the certainty of the efficiency of the medical treatment. So, it seems natural to combine the certainty degrees of rules allowing to determine the level of certainty of the conclusions.

This section continues with a theoretical background on possibilistic logic and ASP. Section 2 introduces possibilistic definite logic programs and their possibilistic models. Section 3 extends our approach to possibilistic stable models for logic programs with default negation. Last, section 4 links our work to some others in the same area and gives some perspectives.

Possibilistic logic. Possibilistic logic, in the necessity-valued case, handles pairs of the form (p, α) where p is a classical logic formula and α is an element of a totally ordered set. The pair (p, α) expresses that the formula p is certain at least to the level α . A possibilistic base is denoted $\Sigma = \{(p_i, \alpha_i)\}_{i \in I}$. Formulas with degree 0 are not explicitly represented in the knowledge base since only beliefs which are somewhat accepted are useful. The higher is the weight,

the more certain is the formula. This degree α is evaluated by a necessity measure and it is not a probability. Thus, numerical values are not an absolute evaluation (like it is in probability theory) but induce a certainty (or confidence) scale. Moreover, let us note that these values are determined by the expert providing the knowledge base or they are automatically given if, as we can imagine, the rules and their confidence degrees result from a knowledge discovery process.

The basic element of possibility theory is the possibility distribution π which is a mapping from Ω , the interpretation set, to the interval $[0, 1]$. $\pi(\omega)$ represents the degree of compatibility of the interpretation ω with the available information (or beliefs) about the real world. By convention, $\pi(\omega) = 0$ means that ω is impossible, and $\pi(\omega) = 1$ means that nothing prevents ω from being the real world (ω is consistent with all the available beliefs, it is a model of Σ). When $\pi(\omega) > \pi(\omega')$, ω is a preferred candidate to ω' for being the real state of the world. From a possibility distribution π , we can define two different ways of rank-ordering formulas of the language. This is obtained using two mappings grading respectively the possibility and the certainty of a formula p : $\Pi(p) = \max\{\pi(\omega) \mid \omega \models p\}$ is the possibility (or consistency) degree which evaluates the extent to which p is consistent with the available beliefs expressed by π [Zadeh, 1978]; $N(p) = 1 - \Pi(\neg p) = 1 - \max\{\pi(\omega) \mid \omega \not\models p\}$ is the necessity (or certainty) degree which evaluates the extent to which p is entailed by the available beliefs.

Moreover, given a base Σ , only the possibility distributions respecting $\forall(p, \alpha) \in \Sigma, N(p) \geq \alpha$ are significant and called *compatible*. A possibility distribution π is said to be the least specific one between all compatible distributions if there is no possibility distribution π' , with $\pi' \neq \pi$, compatible with Σ such that $\forall\omega, \pi'(\omega) \geq \pi(\omega)$. The least specific possibility distribution π_Σ always exists [Dubois *et al.*, 1995] and is defined by:

if ω is a model of Σ , then $\pi_\Sigma(\omega) = 1$
 else $\pi_\Sigma(\omega) = 1 - \max\{\alpha \mid \omega \not\models p, (p, \alpha) \in \Sigma\}$.

Stable Model Semantics. ASP is concerned by different kinds of logic programs and different semantics. In our work we deal with *normal logic programs*, interpreted by *stable model semantics* [Gelfond and Lifschitz, 1988]. We consider given a non empty set of atoms $\mathcal{X}$ that determines the language of the programs¹. A *normal logic program* is a set of rules of the form: $c \leftarrow a_1, \dots, a_n, \text{not } b_1, \dots, \text{not } b_m$, where $n \geq 0, m \geq 0, \{a_1, \dots, a_n, b_1, \dots, b_m, c\} \subseteq \mathcal{X}$. A term like *not b* is called a *default negation*. The intuitive meaning of such a rule is: "if you have all the a_i 's and no b_j 's then you can conclude c ". For such a rule r we use the following notations (extended to a rule set as usual): the positive prerequisites of r , $\text{body}^+(r) = \{a_1, \dots, a_n\}$; the negative prerequisites of r , $\text{body}^-(r) = \{b_1, \dots, b_m\}$; the conclusion of r , $\text{head}(r) = c$ and the positive projection of r , $r^+ = \text{head}(r) \leftarrow \text{body}^+(r)$. If a program P does not contain any default negation (ie: $\text{body}^-(P) = \emptyset$), then P is a *definite logic program* and it has one minimal

Herbrand model $Cn(P)$. The *reduct* P^X of a program P wrt. an atom set X is the definite logic program defined by: $P^X = \{r^+ \mid r \in P, \text{body}^-(r) \cap X = \emptyset\}$ and it is the core of the definition of a *stable model*.

Definition 1.1 [Gelfond and Lifschitz, 1988] *Let P be a normal logic program and S an atom set. S is a stable model of P if and only if $S = Cn(P^S)$.*

Note that a program may have one or many stable models or not at all. In this last case we say that the program is *inconsistent*, otherwise it is *consistent*. When an atom belongs to at least to one stable model of P it is called a *credulous* consequence of P and when it belongs to every stable model of P , it is called a *skeptical* consequence of P .

Let A be an atom set, r be a rule and P be a program (definite or normal). We say that r is *applicable* in A if $\text{body}^+(r) \subseteq A$ and we denote $\text{App}(P, A)$ the subset of P of its applicable rules in A . A satisfies r (or r is satisfied by A), denoted $A \models r$, if r is applicable in $A \Rightarrow \text{head}(r) \in A$. A is *closed* under P if $\forall r \in P, A \models r$. P is *grounded* if it can be ordered as the sequence $\langle r_1, \dots, r_n \rangle$ such that $\forall i, 1 \leq i \leq n, r_i \in \text{App}(P, \text{head}(\{r_1, \dots, r_{i-1}\}))$. $Cn(P)$, the least (Herbrand) model of a definite logic program P , is the smallest atom set closed under P and it can be computed as the least fix-point of the following consequence operator $T_P : 2^{\mathcal{X}} \rightarrow 2^{\mathcal{X}}$ such that: $T_P(A) = \text{head}(\text{App}(P, A))$.

By this way, we can establish the next result that clarifies the links between the least model A of a program P and the rules producing it. We see that A is underpinned by a set of applicable rules, $\text{App}(P, A)$, that satisfies a stability condition and that is grounded. These two features will be used in the sequel to define the core of our work: a possibility distribution over atom sets induced by a definite logic program.

Lemma 1.1 *Let P be a definite logic program and A be an atom set,*

*A is the least
Herbrand model of P* $\iff \begin{cases} A = \text{head}(\text{App}(P, A)) \\ \text{App}(P, A) \text{ is grounded} \end{cases}$

2 Possibilistic Definite Logic Programs

We consider given a finite set of atoms $\mathcal{X}$ and a finite, totally ordered set of necessity values $\mathcal{N} \subseteq]0, 1]$. Then, a *possibilistic atom* is a pair $p = (x, \alpha) \in \mathcal{X} \times \mathcal{N}$. We denote by $p^* = x$ the classical projection of p and by $n(p) = \alpha$ its necessity degree. These notations can also be extended to a *possibilistic atom set* (p.a.s.) A that is a set of possibilistic atoms where every atom x occurs at most one time in A .

A *possibilistic definite logic program* (p.d.l.p.) is a set of *possibilistic rules* of the form: $r = (c \leftarrow a_1, \dots, a_n, \alpha)$ $n \geq 0, \{a_1, \dots, a_n, c\} \subseteq \mathcal{X}, \alpha \in \mathcal{N}$. The *classical projection* of the possibilistic rule is $r^* = c \leftarrow a_1, \dots, a_n$. $n(r) = \alpha$ is a necessity degree representing the certainty level of the information described by r . If R is a set of possibilistic rules, then $R^* = \{r^* \mid r \in R\}$ is the definite logic program obtained from P by forgetting all the necessity values.

Let us recall that a possibilistic logic base is a compact representation of the possibility distribution defined on interpretations representing the information. Indeed, the treatment of

¹In the sequel, if $\mathcal{X}$ is not explicitly given, it is supposed to be the set of all atoms occurring in the considered program.

the base in a syntactic way (in terms of formulas and necessity degrees) leads to the same results as the treatment done in a semantic way (in terms of interpretations and possibility distribution). In our framework, the same situation occurs as it is shown in the next two subsections. Firstly, we define a semantic handling of a program that is defined in term of a possibility distribution over all atom sets. Secondly, we provide a syntactic deduction process that leads to the same results as the ones given by the possibility distribution.

2.1 Model Theory for Possibilistic Definite Logic Programs

From a possibilistic definite logic program P , we can determine, as it is done in possibilistic logic, some possibility distributions defined on all the sets in $2^{\mathcal{X}}$ and that are compatible with P . Let us note that there is a correspondence between interpretations in propositional logic and atom sets in ASP. Like in possibilistic logic, the possibility degree of an atom set is determined by the necessity degrees of the rules of the program that are not satisfied by this set. The satisfiability of a rule r is based on its applicability wrt. an atom set A , so $A \models r$ iff $\text{body}^+(r) \subseteq A$ and $\text{head}(r) \notin A$ (see section 1). But, we have to notice that the contradiction of a rule is not enough to determine the possibility degree of a set since, in ASP, it is important to take into account the groundedness and stability notions (see lemma 1.1). Firstly, the set $A = \{a, b\}$ satisfies every rule in $P = \{a \leftarrow b., b \leftarrow a.\}$, but it is not the least model of P because the groundedness is not satisfied. Secondly, the set $A' = \{a, b, d\}$ satisfies every rule in $P' = \{a., b \leftarrow a., d \leftarrow c.\}$ but it is not the least model of P' because d can not be produced by any rule from P' applicable in A' . In these two cases, the possibility of A and A' must be 0 since they can not be the least model at all, even if they satisfy every rule in their respective associated programs.

Definition 2.1 Let P be a p.d.l.p. and $\pi : 2^{\mathcal{X}} \rightarrow [0, 1]$ be a possibility distribution. π is compatible with P if

$$\begin{cases} A \not\subseteq \text{head}(\text{App}(P^*, A)) \Rightarrow \pi(A) = 0 \\ \text{App}(P^*, A) \text{ is not grounded} \Rightarrow \pi(A) = 0 \\ A \text{ is the least model of } P^* \Rightarrow \pi(A) = 1 \\ \text{otherwise } \pi(A) \leq 1 - \max_{r \in P} \{n(r) \mid A \not\models r^*\} \end{cases}$$

The necessity degree attached to each rule defines only a lower bound (and not an exact value) of the certainty of the rule. So, as recalled in section 1, many possibility distributions are compatible with these degrees. But, we are only interested by the least informative one, that is the *least specific* one, whose characterization is given below.

Proposition 2.1 Let P be a p.d.l.p. then $\pi_P : 2^{\mathcal{X}} \rightarrow [0, 1]$ defined by

1. if $A \not\subseteq \text{head}(\text{App}(P^*, A))$ then $\pi_P(A) = 0$
2. if $\text{App}(P^*, A)$ is not grounded then $\pi_P(A) = 0$
3. otherwise
 - if $\forall r \in P, A \models r^*$ then $\pi_P(A) = 1$
 - otherwise $0 \leq \pi_P(A) < 1$ and $\pi_P(A) = 1 - \max_{r \in P} \{n(r) \mid A \not\models r^*\}$

is the least specific possibility distribution.

The definition of π_P ensures that it is compatible with P . The third item ranks the sets which may be solutions with respect to the weights of the falsified rules.

Proposition 2.2 Let P be a p.d.l.p. and $A \subseteq \mathcal{X}$ be an atom set, then

1. $\pi_P(A) = 1 \iff A = Cn(P^*)$
2. $A \supset Cn(P^*) \Rightarrow \pi_P(A) = 0$
3. $Cn(P^*) \neq \emptyset \Rightarrow \pi_P(\emptyset) = 1 - \max_{r \in P} \{n(r) \mid \emptyset \not\models r^*\}$

Now, we can give the definition of inference that is, in the framework of ASP, the evaluation of the necessity degree of each atom of the universe.

Definition 2.2 Let P be a p.d.l.p. and π_P the least specific possibility distribution compatible with P , we define the two dual possibility and necessity measures such that:

$$\begin{aligned} \Pi_P(x) &= \max_{A \in 2^{\mathcal{X}}} \{\pi_P(A) \mid x \in A\} \\ N_P(x) &= 1 - \max_{A \in 2^{\mathcal{X}}} \{\pi_P(A) \mid x \notin A\} \end{aligned}$$

$\Pi_P(x)$ gives the level of consistency of x with respect to the p.d.l.p. P and $N_P(x)$ evaluates the level at which x is inferred from the p.d.l.p. P . This is closely related to the definitions of possibilistic logic. For instance, whenever an atom x belongs to the least model of the classical program its possibility is total.

Proposition 2.3 Let P be a p.d.l.p., $Cn(P^*)$ the least model of P^* and $x \in \mathcal{X}$ then :

1. $x \in Cn(P^*) \Rightarrow \Pi_P(x) = 1$ and $x \notin Cn(P^*) \Rightarrow \Pi_P(x) = 0$
2. $x \notin Cn(P^*) \iff N_P(x) = 0$
3. $x \in Cn(P^*) \Rightarrow N_P(x) = \min_{A \subseteq Cn(P^*)} \{\max_{r \in P} \{n(r) \mid A \not\models r^*\} \mid x \notin A\}$
4. Let P' be a p.d.l.p., $P \subseteq P' \Rightarrow N_P(x) \leq N_{P'}(x)$.

Furthermore, the necessity measure allows us to introduce the following definition of a *possibilistic model* of a p.d.l.p.

Definition 2.3 Let P be a possibilistic definite logic program, then the set

$$\Pi M(P) = \{(x, N_P(x)) \mid x \in \mathcal{X}, N_P(x) > 0\}$$

is its possibilistic model.

Proposition 2.4 Let P be a p.d.l.p. then : $(\Pi M(P))^*$ is the least model of P^* .

Example 2.1 Let us take the following program $P = \{(a., 0.8), (b \leftarrow a., 0.6), (d \leftarrow a., 0.5), (d \leftarrow c., 0.9)\}$. The least specific possibility distribution induced by P is null for every atom set included in $\{a, b, c, d\}$ except for $\pi_P(\emptyset) = 0.2$, $\pi_P(\{a\}) = 0.4$, $\pi_P(\{a, b\}) = 0.5$, $\pi_P(\{a, d\}) = 0.4$ and $\pi_P(\{a, b, d\}) = 1$ (the least model of P^*). So, we can compute the possibility of each atom: $\Pi_P(a) = \Pi_P(b) = \Pi_P(d) = 1$ and $\Pi_P(c) = 0$ and its certainty in term of necessity degree: $N_P(a) = 0.8$, $N_P(b) = 0.6$, $N_P(c) = 0$, $N_P(d) = 0.5$. Thus, $\Pi M(P) = \{(a, 0.8), (b, 0.6), (d, 0.5)\}$ is the possibilistic model of P .

2.2 Fix-point Theory for Possibilistic Definite Logic Programs

Definitions exposed in this subsection are closely related to what can be found in [Dubois *et al.*, 1991]. But here, we adopt an ASP point of view and thus we use atom sets instead of classical interpretations since the underlying possibility distribution is defined on atom sets.

Definition 2.4 Consider $\mathcal{A} = 2^{\mathcal{X} \times \mathcal{N}}$ the finite set of all p.a.s. induced by $\mathcal{X}$ and $\mathcal{N}$. $\forall A, B \in \mathcal{A}$, we define:

$$\begin{aligned} A \sqcap B &= \{(x, \min\{\alpha, \beta\}), (x, \alpha) \in A, (x, \beta) \in B\} \\ A \sqcup B &= \{(x, \alpha) \mid (x, \alpha) \in A, x \notin B^*\} \\ &\quad \cup \{(x, \beta) \mid x \notin A^*, (x, \beta) \in B\} \\ &\quad \cup \{(x, \max\{\alpha, \beta\}) \mid (x, \alpha) \in A, (x, \beta) \in B\} \\ A \sqsubseteq B &\iff \begin{cases} A^* \subseteq B^*, \text{ and } \forall a, \alpha, \beta, \\ (a, \alpha) \in A \wedge (a, \beta) \in B \Rightarrow \alpha \leq \beta \end{cases} \end{aligned}$$

Proposition 2.5 $\langle \mathcal{A}, \sqsubseteq \rangle$ is a complete lattice.

Definition 2.5 Let $r = (c \leftarrow a_1, \dots, a_n, \alpha)$ be a possibilistic rule and A be a p.a.s.,

- r is α -applicable in A if $\text{body}(r^*) = \emptyset$
- r is β -applicable in A if $\beta = \min\{\alpha, \alpha_1, \dots, \alpha_n\}$, $\{(a_1, \alpha_1), \dots, (a_n, \alpha_n)\} \subseteq A$,²
- r is 0-applicable otherwise.

And, for a given p.d.l.p. P and an atom x , $\text{App}(P, A, x) = \{r \in P \mid \text{head}(r^*) = x, r \text{ is } \nu\text{-applicable in } A, \nu > 0\}$

The applicability degree of a rule r captures the certainty of the inference process realized when r is applied wrt. a p.a.s. A . If the body of r is empty, then r is applicable with its own certainty. If the body of r is not verified (not satisfied by A), then r is not at all applicable. Otherwise, the applicability level of r depends on the certainty level of the propositions inducing its groundedness and on its own necessity degree. Firstly, the necessity degree of a conjunction (the rule body) is the minimal value of the necessity values of subformulae (atoms) involved in it. Secondly, the certainty of a rule application is the minimal value between the rule certainty and the certainty of the rule body. This approach is similar to the resolution principle in possibilistic logic [Dubois *et al.*, 1995].

Definition 2.6 Let P be a p.d.l.p. and A be a possibilistic atom set. The immediate possibilistic consequence operator ΠT_P maps a p.a.s. to another one by this way: $\Pi T_P(A) =$

$$\left\{ \begin{array}{l} (x, \delta) \mid x \in \text{head}(P^*), \text{App}(P, A, x) \neq \emptyset, \\ \delta = \max_{r \in \text{App}(P, A, x)} \{\nu \mid r \text{ is } \nu\text{-applicable in } A\} \end{array} \right\}$$

then the iterated operator ΠT_P^k is defined by

$$\Pi T_P^0 = \emptyset \quad \text{and} \quad \Pi T_P^{n+1} = \Pi T_P(\Pi T_P^n), \forall n \geq 0.$$

Here, we can remark that if one conclusion is obtained by different rules, its certainty is equal to the greatest certainty with which it is obtained in each case (operator max). Again, it is in accordance with possibilistic resolution principle.

Proposition 2.6 Let P be a p.d.l.p., then ΠT_P has a least fix-point $\sqcup_{n \geq 0} \Pi T_P^n(\emptyset)$ that we called the set of possibilistic consequences of P and we denote it by $\Pi Cn(P)$.

²For two p.a.s. A and B , $A \subseteq B$ means the classical set inclusion and has not to be confused with the order $\sqsubseteq$ of def. 2.4.

Example 2.2 Let P be the p.d.l.p. of example 2.1, then $\Pi T_P^0(\emptyset) = \emptyset$, $\Pi T_P^1(\emptyset) = \{(a, 0.8)\}$, $\Pi T_P^2(\emptyset) = \{(a, 0.8), (b, 0.6), (d, 0.5)\}$, $\Pi T_P^3(\emptyset) = \{(a, 0.8), (b, 0.6), (d, 0.5)\} = \Pi T_P^k(\emptyset), \forall k > 3$. So, $\Pi Cn(P) = \{(a, 0.8), (b, 0.6), (d, 0.5)\}$.

Theorem 2.1 Let P be a p.d.l.p., then $\Pi Cn(P) = \Pi M(P)$.

As illustrated and formalized above, our operator ΠT_P can be used to compute the possibilistic model of a p.d.l.p. This result shows the equivalence between the syntactic and semantic approaches in our framework, whereas in [Dubois *et al.*, 1991] only a syntactic treatment of a p.d.l.p. is proposed.

3 Possibilistic Normal Logic Programs

3.1 Possibilistic Normal Logic Programs and Possibilistic Stable Models

Here, we extend our proposal to non-monotonic reasoning by allowing default negation in programs and we formalize the notion of *possibilistic stable model*. It extends the stable model semantics by taking into account the necessity degree in the rules of a given *possibilistic normal logic program* (p.n.l.p.). Such a program is a finite set of rules of the form : $(c \leftarrow a_1, \dots, a_n, \text{not } b_1, \dots, \text{not } b_m, \alpha)$ $n \geq 0, m \geq 0$ for which we just have to precise that $\forall i, b_i \in \mathcal{X}$, all the rest being the same as for a p.d.l.p. (see the beginning of section 2).

As in the classical case without necessity value, we need to define what is the reduction of a program before introducing the *possibilistic stable model semantics*.

Definition 3.1 Let P be a p.n.l.p. and A be an atom set. The possibilistic reduct of P wrt. A is the p.d.l.p. $P^A = \{((r^*)^+, n(r)) \mid r \in P, \text{body}^-(r) \cap A = \emptyset\}$.

Definition 3.2 Let P be a p.n.l.p. and S a p.a.s. S is a *possibilistic stable model* of P if and only if $S = \Pi Cn(P^{(S^*)})$.

By analogy with classical normal logic programs (without necessity values attached to rules) we say that a p.n.l.p. P is *consistent* if P has at least one possibilistic stable model. Otherwise P is said to be *inconsistent*. Furthermore when a possibilistic atom belongs at least to one possibilistic stable model of P it is called a *credulous* possibilistic consequence of P and when it belongs to every possibilistic stable model of P , it is called a *skeptical* possibilistic consequence of P .

Example 3.1 We are now able to represent the example of the introductory section with some levels of certainty by $P = \{(dr1 \leftarrow di1, \text{not } dr2, 1), (dr2 \leftarrow di2, \text{not } dr1, 1), (c1 \leftarrow dr1, di1, 0.7), (c2 \leftarrow dr2, di2, 0.3), (di1, 0.9), (di2, 0.7)\}$. The two first rules express that for each disease we have an appropriate drug, these two drugs are incompatible, but their appropriateness is fully certain. The third (resp. fourth) rule expresses that drug $dr1$ (resp. $dr2$) has an efficiency quasi certain (resp. little certain) for the disease $di1$ (resp. $di2$). The two last rules (some observations) represent that the diagnosis of the disease $di1$ (resp. $di2$) is quasi certain (resp. almost certain). Then, the two possibilistic stable models of P are $\{(di1, 0.9), (di2, 0.7), (dr1, 0.9), (c1, 0.7)\}$ and $\{(di1, 0.9), (di2, 0.7), (dr2, 0.7), (c2, 0.3)\}$. So, the doctor

can observe that he has an alternative. On one hand, he can give the drug *dr1* and the patient will be almost certainly cured of the disease *di1*. On the other hand, he can give the drug *dr2* and the patient will be cured of the disease *di2* with a low certainty degree. However, if the doctor considers that disease *di2* is very serious, maybe he will choose the drug *dr2* even if the degree is lower. That is why it is interesting to keep the two stable models in which every conclusion is weighted with a certainty degree.

Proposition 3.1 *Let P be a p.n.l.p.*

1. *Let A be a possibilistic stable model of P and $\alpha \in \mathcal{N}, \alpha > 0$, then $(x, \alpha) \in A \iff \alpha = N_{PA^*}(x)$.*
2. *Let S be a stable model of P^* , then $\{(x, N_{PS}(x)) \mid x \in \mathcal{X}, N_{PS}(x) > 0\}$ is a possibilistic stable model of P .*
3. *Let A be a possibilistic stable model of P , then A^* is a stable model of P^* .*

This result shows that there is a one-to-one mapping between the possibilistic stable models of a p.n.l.p P and the stable models of its classical part P^* . It shows that the decision problem of existence of a p.s.m. for a p.n.l.p. stays in the same complexity class as the decision problem of existence of a stable model for a normal logic program. So, it is an NP-complete problem. Furthermore, it indicates an easy way to implement a system able to compute a p.s.m. of a p.n.l.p P . First, one can use a software (as *Smodels* [Niemelä and Simons, 1997] or *DLV* [Eiter et al., 1998]) to compute the stable models of P^* . Second, for every found stable model A , our operator ΠT_{PA} can be used to compute (in polynomial time) the corresponding p.s.m. $\Pi Cn(P^A)$ of P .

3.2 Possibility Distribution defined upon a Normal Logic Program

In the previous definition 3.2 we have proposed a syntactic way to compute the possibilistic stable models of a p.n.l.p. by using the fix-point operator ΠCn defined for a p.d.l.p. Now, we examine the semantics that can be given to this framework by defining a possibility distribution induced by the necessity values associated to each normal rules. This distribution $\tilde{\pi}$ over all the atom sets (i.e. over $2^{\mathcal{X}}$) has to reflect the ability of every atom set to be a stable model of P^* and that is achieved by the next definitions.

Definition 3.3 *Let P be a p.n.l.p. and A be an atom set, then $\tilde{\pi}_P$ is the possibility distribution defined by $\tilde{\pi}_P : 2^{\mathcal{X}} \rightarrow [0, 1]$ and respecting: $\forall A \in 2^{\mathcal{X}}, \tilde{\pi}_P(A) = \pi_{PA}(A)$.*

One can observe that definition of $\tilde{\pi}_P$ can be paraphrased by: "the possibility for an atom set A to be a stable model of P is its possibility to be the least model of the program P reduced by A ". This definition is natural wrt. the definition 1.1 of a stable model and is justified by the next result.

Proposition 3.2 *Let P be a p.n.l.p. and $A \in 2^{\mathcal{X}}$ be an atom set, then $\tilde{\pi}_P(A) = 1 \iff A$ is a stable model of P^* .*

Example 3.2 *The possibility distribution induced by the p.n.l.p. $P = \{(a., 1), (b., 1), (c \leftarrow a, \text{not } d., 0.4), (d \leftarrow b, \text{not } c., 0.8), (e \leftarrow c., 0.6), (e \leftarrow d., 0.5)\}$ is null for every atom set included in $\{a, b, c, d, e\}$ except for $\tilde{\pi}_P(\{a, b\}) =$*

0.2, $\tilde{\pi}_P(\{a, b, c\}) = 0.4$, $\tilde{\pi}_P(\{a, b, d\}) = 0.5$ and $\tilde{\pi}_P(\{a, b, c, e\}) = \tilde{\pi}_P(\{a, b, d, e\}) = 1$ ($\{a, b, c, e\}$ and $\{a, b, d, e\}$ are the two stable models of P^).*

Definition 3.4 *Let P be a p.n.l.p. and $\tilde{\pi}_P$ its associated possibility distribution, we define the two dual possibility and necessity measures such that:*

$$\begin{aligned}\tilde{\Pi}_P(x) &= \max_{A \in 2^{\mathcal{X}}} \{\tilde{\pi}_P(A) \mid x \in A\} \\ \tilde{N}_P(x) &= 1 - \max_{A \in 2^{\mathcal{X}}} \{\tilde{\pi}_P(A) \mid x \notin A\}.\end{aligned}$$

Proposition 3.3 *Let P be a consistent p.n.l.p. and $x \in \mathcal{X}$,*

1. *x is a credulous consequence of $P^* \Rightarrow \tilde{\Pi}_P(x) = 1$*
2. *x is not a skeptical consequence of $P^* \iff \tilde{N}_P(x) = 0$*

Let us remark that if an atom x is not a credulous consequence of P then it does not necessarily imply that $\tilde{\Pi}_P(x) = 0$ as it is the case for an atom that is not a consequence of a p.d.l.p. (see proposition 2.3 item 1). For instance, $P = \{(a., 0.6), (b \leftarrow \text{not } a., 0.7)\}$ has only one p.s.m. $\{(a, 0.6)\}$ so b is not a possibilistic credulous consequence of P . But, the possibility distribution is $\tilde{\pi}_P(\emptyset) = 0.3, \tilde{\pi}_P(\{a\}) = 1, \tilde{\pi}_P(\{b\}) = 0.4$ and $\tilde{\pi}_P(\{a, b\}) = 0$ so $\tilde{\Pi}_P(b) = 0.4$. This is because, with this program, b is not completely impossible. In fact, b is not a credulous consequence only because a blocks the applicability of the rule concluding b . So, in other words if we have a then we can not have b , but if a is absent then we can have b . Since, the certainty of a is only 0.6, so the possibility of b is naturally 0.4.

4 Discussion and Conclusion

In this work, we have proposed a way to deal with logic programs taking into account a level of certainty for each rule. First, we have given two equivalent ways, a semantic and a syntactic one, for dealing with possibilistic definite logic programs. Then, we have shown how to extend our framework to possibilistic normal logic programs in order to obtain what we called possibilistic stable models. In addition to this work, we have described in [Nicolas et al., 2005] an application of our proposal to deal with inconsistent normal logic programs.

There are alternative formalisms for extending non-monotonic reasoning with qualitative or quantitative weights. In [Nerode et al., 1997], annotated non-monotonic rule systems are introduced to affect on every piece of knowledge a weight representing a probability, an uncertainty measure, a time, a place, ... For instance for a logic program, it leads to rules like $(c, 0.4) \leftarrow (a, 0.5), (\text{not } b, 0.7)$. This approach is different to our since we put weights on rules and not on each atom in the rules. Another divergence is the meaning of the weights : in our work we deal with uncertainty while they develop a formalism in which the semantic of weights is not established a priori. In [Damasio and Pereira, 2001a; 2001b; Dubois et al., 1991; Benferhat et al., 1993; Loyer and Straccia, 2003; Alsinet and Godo, 2000] the reader can find different propositions about multi-valued or probabilistic logic programs, about possibilistic definite logic programs, about levels of certainty ranking atoms, but not rules, involved in a non-monotonic reasoning, ... But, none of these works describes a formalism dealing with uncertainty in a logic program with default negation by means of possibilistic theory,

both in a semantic and a syntactic ways. The closest work of ours can be found in [Wagner, 1997]. But the approach is to reconstruct possibilistic logic by defining a multi-valued interpretation and a corresponding satisfaction relation and forgetting the notion of possibility distribution. Then, he is not able to provide any result about the possibility of an atom set to be a stable model. Moreover, the necessity (certainty) of a conclusion is not correlated to the possibility of the atom set. So, we think that this approach is far from the spirit of possibilistic logic.

In possibilistic logic, the necessity degrees are commonly interpreted as preferences between formulas, the more certain is a formula, the more preferred it is. Since there is a lot of works dealing with preferences between rules in non-monotonic reasoning [Delgrande *et al.*, 2004], it is interesting to analyze our work when we consider that the necessity degrees on rules determine a preference order. If we look only on the area of ASP, most of these works use the preferences expressed between the rules to make a choice between the different stable models to keep only the "preferred" ones. In other words, the priorities between the rules do not evaluate a certainty degree of the rules but are a tool to choice between contradicting rules. This differs from our work because, when several stable models exist, we compute the certainty of the propositions with respect to each stable model. But we do not try to eliminate some stable models since we consider they are alternate solutions. Nevertheless, a future work analyzing in depth the links between our proposal and the preference handling in ASP is envisaged, in particular with respect to the two general principles stated in [Brewka and Eiter, 1999].

Some direct, and without theoretical difficulties, continuation of this work is to precise the handling of strong negation in possibilistic normal programs. Then, we also plan to develop a system to compute possibilistic stable models by interfacing it with existing systems able to compute classical stable models. Furthermore, we will try to extend our approach to disjunctive logic programs and answer set semantics [Gelfond and Lifschitz, 1991].

References

- [Alsinet and Godo, 2000] T. Alsinet and L. Godo. A complete calculus for possibilistic logic programming with fuzzy propositional variables. In C. Boutilier and M. Goldszmidt, editors, *UAI '00: Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence*, pages 1–10, Stanford University, Stanford, California, USA, 2000. Morgan Kaufmann.
- [Benferhat *et al.*, 1993] S. Benferhat, D. Dubois, and H. Prade. Possibilistic logic: From nonmonotonicity to logic programming. In M. Clarke, R. Kruse, and S. Moral, editors, *European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, volume 747 of *LNCS*, pages 17–24. Springer, 1993.
- [Brewka and Eiter, 1999] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109(1-2):297–356, 1999.
- [Damasio and Pereira, 2001a] C. V. Damasio and L. M. Pereira. Antitonic logic programs. In T. Eiter, W. Faber, and M. Truszczynski, editors, *International Conference on Logic Programming and NonMonotonic Reasoning*, volume 2173 of *LNCS*, pages 379–392. Springer, 2001.
- [Damasio and Pereira, 2001b] C. V. Damasio and L. M. Pereira. Monotonic and residuated logic programs. In S. Benferhat and P. Besnard, editors, *European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, volume 2143 of *LNCS*, pages 748–759. Springer, 2001.
- [Delgrande *et al.*, 2004] J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(2):308–334, 2004.
- [Dubois and Prade, 1998] D. Dubois and H. Prade. Possibility theory: qualitative and quantitative aspects. In Ph. Smets, editor, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 1, pages 169–226. Kluwer Academic Press, 1998.
- [Dubois *et al.*, 1991] D. Dubois, J. Lang, and H. Prade. Towards possibilistic logic programming. In K. Furukawa, editor, *International Conference on Logic Programming*, pages 581–595, Cambridge, MA, 1991. MIT Press.
- [Dubois *et al.*, 1995] D. Dubois, J. Lang, and H. Prade. Possibilistic logic. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3, pages 439–513. Oxford University Press, 1995.
- [Eiter *et al.*, 1998] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The kr system dlvr: progress report, comparisons and benchmarks. pages 406–417. Morgan Kaufmann Publishers, 1998.
- [Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *International Conference on Logic Programming*, pages 1070–1080. The MIT Press, 1988.
- [Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3-4):363–385, 1991.
- [Loyer and Straccia, 2003] Y. Loyer and U. Straccia. Default knowledge in logic programs with uncertainty. In *International Conference on Logic Programming*, volume 2916 of *LNCS*, pages 466–480, Mumbai, India, 2003. Springer.
- [Nerode *et al.*, 1997] A. Nerode, J. B. Remmel, and V. S. Subrahmanian. Annotated nonmonotonic rule systems. *Theoretical Computer Science*, 171(1-2):111–146, 1997.
- [Nicolas *et al.*, 2005] P. Nicolas, L. Garcia, and I. Stéphan. A possibilistic inconsistency handling in answer set programming. In *European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, *LNCS*, Barcelona, Spain, 2005. Springer.
- [Niemelä and Simons, 1997] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *International Conference on Logic Programming and NonMonotonic Reasoning*, volume 1265 of *LNCS*, pages 420–429. Springer, 1997.
- [Wagner, 1997] G. Wagner. A logical reconstruction of fuzzy inference in databases and logic programs. In *Proc. of Int. Fuzzy Set Association World Congress (IFSA)*, Prague, Czech, 1997.
- [Zadeh, 1978] L.A. Zadeh. Fuzzy sets as a basis for a theory of possibility. In *Fuzzy Sets and Systems*, volume 1, pages 3–28. 1978.

Game-Tree Search with Combinatorially Large Belief States

Austin Parker, Dana Nau, V.S. Subrahmanian

Department of Computer Science, University of Maryland

College Park, MD, 20742, USA

Email: {austinjp,nau,vs}@cs.umd.edu

Abstract

In games such as kriegspiel chess (a chess variant where players have no direct knowledge of the opponent's pieces' locations) the belief state's sizes dwarf those of other partial information games like bridge, scrabble, and poker—and there is no easy way to generate states satisfying the given observations. We show that statistical sampling approaches can be developed to do well in such games.

We show that it is not necessary for the random sample to consist only of game boards that satisfy each and every one of a player's observations. In fact, we win 24% more often by beginning with such completely consistent boards and gradually switching (as the game progressed) to boards that are merely consistent with the latest observation.

This surprising result is explained by noting that as the game progresses, a board that is consistent with the last move becomes more and more likely to be consistent with the entire set of observations, even if we have no idea what sequence of moves might have actually generated this board.

1 Introduction

One of the biggest difficulties in using game-tree search algorithms in imperfect-information games is that the game trees have huge branching factors. Usually the *belief-state size* (the number of current states that are consistent with everything a player knows about the game) is combinatorially large. Hence the number of moves that the opponent *might* be able to make, and the number of states that these moves might lead to, are also quite large.

The best known way to circumvent this problem is to do statistical sampling: to generate a set of random game boards consistent with the current belief state, do a perfect-information game-tree search on each of those game boards, and average the minimax values. This has worked well in games such as bridge [Smith *et al.*, 1998; Ginsberg, 1999], scrabble [Sheppard, 2002], and poker [Billings *et al.*, 2003].

The success of statistical sampling in the above games relates, at least in part, to several conditions that do not hold in various other imperfect-information games, such as

kriegspiel chess, Civilization, StarCraft, Quake3, Counter-Strike, and FreeCiv. More specifically:

- *Belief-state size.* Bridge, poker, and scrabble have relatively small belief states. At the start of a bridge hand, the size of each player's belief state is about 10,000,000 after the dummy's cards have been exposed; and this number decreases exponentially as the game proceeds. Although 10,000,000 may seem like a large number, it is dwarfed by the belief-state sizes in several other games. For example, in kriegspiel chess,¹ the size of each belief state at mid-game is well above 10^{13} ; and in the *small* world map for FreeCiv, the belief-state size is about 10^{30} .
- *Uncertainty model.* The uncertainty in bridge, poker, and scrabble is due to external factors that can easily be modeled stochastically: the random deal of the cards in bridge or poker, and the random choice of a tile in Scrabble. Thus it is easy to tell whether or not a state s is consistent with a belief state b , and to assign a probability to s given b .
In games such as kriegspiel chess, the uncertainty arises from lack of observability of the opponent's actions. This uncertainty has no simple stochastic model. Instead, telling whether a state s is consistent with the current belief state b means checking whether there is a *history* (i.e., a sequence of moves) that is consistent with b and leads to s . In the average case, this takes exponential time.
- *Simplifying approximations.* In bridge and poker, usually the statistical sampling is not done on the game itself, but on a simplified approximation in which the state space and belief states are much smaller. In bridge, the approximation is done by treating various sets of states as if they were equivalent, a technique that was first used in the game of sprouts [Applegate *et al.*, 1991]. In poker, a linear-programming approximation has been used [Billings *et al.*, 2003]. In kriegspiel chess, it is unclear how or whether such an approximation could be constructed.

To summarize, our objective is to determine how to make game-tree search work successfully in imperfect-information games that have huge belief states, no simple stochastic model of uncertainty, and no obvious way to construct a simplified approximation of the game. Our results are:

¹Kriegspiel chess is like ordinary chess except that neither player can see the other's pieces. Section 3 gives a fuller description.

```

procedure Choose-move( $S$ )
 $M \leftarrow \{\text{moves applicable to states in } S\}$ 
for every  $s \in S$  and every  $m \in M$  do
 $v_{s,m} \leftarrow \text{Evaluate-board}(\gamma(s, m))$ 
return  $\text{argmax}_{m \in M} \sum_{s \in S} v_{s,m} P(s)$ 

```

Figure 1: Abstract statistical-sampling algorithm for move evaluation. Evaluate-board is a perfect-information search algorithm such as alpha-beta.

1. We describe several algorithms for generating the random sample of game boards used in the tree search. AOSP generates game boards that are consistent with the entire sequence O of observations that a player has made during the game. LOS only requires consistency with the last observation o_i . HS behaves like AOSP at the beginning of the game, but as the game progresses it gradually switches over to behaving like LOS.
2. We analyze the performance of our algorithms theoretically. Surprisingly, our analysis suggests that there are cases in which LOS will play well and HS will do better than AOSP.
3. Our experimental tests in the game of kriegspiel chess confirm our analysis' hypotheses. In our experiments, Timed LOS (LOS with a time limit) did much better than random play, Timed AOSP did much better than Timed LOS; and Timed HS did much better than Timed AOSP.

2 Game-Tree Search with Statistical Sampling

To describe how statistical sampling is used for game-tree search in imperfect-information games, we need an abstract model for the kinds of games where it is used. For simplicity, we assume the game is zero-sum and there are two players p_1, p_2 who move in alternation.

As the game progresses, the players' moves will generate a sequence of states $S_i = \langle s_0, s_1, \dots \rangle$ called the *game history*. At each state s_i , each player p_j will be able to make an observation o_{ij} of s_i ; usually o_{ij} will include complete information about p_j 's position and partial information about the other player's position. At s_i , player p_j 's *observation history* is $O_{ij} = \langle o_{1j}, o_{2j}, \dots, o_{ij} \rangle$, and p_j 's *belief state* is $b_{ij} = \{\text{all states that satisfy } O_{ij}\}$.

Our sampling algorithms will be based on the following properties of a state s : s is *last-observation consistent* if it is consistent with o_{ij} , and *all-observation consistent* if it is consistent with O_{ij} .

Figure 1 shows an abstract version of statistical game-tree search. S is the sample set of states, $\gamma(s, m)$ is the state produced by performing move m in state s , Evaluate-board is a perfect-information game-tree-search algorithm such as alpha-beta, and P is a probability distribution over the states in S . Some additional code must be added to handle the case where a move m is applicable to some states but not others; this code tends to be game-specific and we discuss it further in Section 4.

We now can define four different sampling algorithms that provide input for Choose-move. In each case, k is the de-

sired number of states in the statistical sample, and i is how many moves the players have played so far.

- **LOS (Last Observation Sampling)** If there are fewer than k last-observation consistent states, then let S contain all of them; otherwise let S contain k such states chosen at random. Return Choose-move(S).
- **AOS (All Observation Sampling)**: Let $S = \emptyset$. Generate a random history $\langle s_1, \dots, s_i \rangle$. If the history satisfies O_{ij} , then add s_i to S . Do this repeatedly until $|S| = k$. Return Choose-move(S).
- **AOSP (All Observation Sampling with Pool)**: AOSP returns both a move and a set of states (a pool) for p_j to use as input to AOSP next move. Every state in the pool is to be consistent with O_{ij} , though we do not assume that all such states are in the pool. Let S_0 be the pool AOSP returned last time, and $M = \{\text{all of the other player's possible responses to } p_j\text{'s last move}\}$. Let $S_1 = \{\gamma(s, m) \mid s \in S_0, m \in M, m \text{ is applicable to } s, \text{ and } \gamma(s, m) \text{ satisfies } O_{ij}\}$. If $|S_1| < k$, then let $S_2 = S_1$; otherwise let S_2 contain k states chosen at random from S_1 . Let $m = \text{Choose-move}(S_2)$. Return $(m, \{\gamma(s, m) \mid s \in S_1\})$.
- **HS (Hybrid Sampling)**: Like AOSP, HS returns a move and a set of states. Compute S_1 and S_2 same as in AOSP. If $|S_2| < k$ then let S_3 be a set of $k - |S_2|$ random last-observation consistent states; otherwise $S_3 = \emptyset$. Let $m = \text{Choose-move}(S_2 \cup S_3)$. Return $(m, \{\gamma(s, m) \mid s \in S_1\})$.

Analyzing the performance of these algorithms is impossible without making simplifying assumptions, but there is more than one set of assumptions one might make. Below we do two analyses, based on two different sets of assumptions. The differing assumptions lead to differing conclusions about which algorithm will do better.

Analysis 1: Suppose each state has exactly b children, for some constant b . Suppose that we know all of p_j 's moves but not the other player's moves. If the number of states is very large (e.g., 10^{13} or 10^{30} as described earlier), then during the early stages of the game, the number of states grows exponentially, with roughly $b^{i/2}$ possible states at the i 'th move. Suppose that for each state s where it is the other player's move, the observation history O_{ij} eliminates, on the average, some fraction $1/c$ of that player's possible moves, where $c > 1$. Then the number of possible states at the i 'th move given O_{ij} is $(b/c)^{i/2}$. Thus the probability of any individual state at depth i being consistent with O_{ij} is $(1/c)^{i/2}$, which approaches 0 at an exponential rate as i increases.

Thus, if the game continues to grow as a tree with a branching factor of b , then our analysis suggests the following:

- AOS will be computationally intractable. The probability of a random history being consistent with b_i is likely to be something like $1/c^{i/2}$. Thus, in order to produce a sample of size k , AOS will probably need to generate $k \cdot c^{i/2}$ histories, taking exponential time.
- AOSP is much more efficient computationally than AOS, but its sample set S_2 will decrease in size as the game progresses. The probability of a state s 's successors being

consistent with b_i is $1/c$, since s is already known to be consistent with b_{i-1} . Hence the amount of time needed to generate k states for S_2 does not increase as quickly as it would if we generated the states from random histories like AOS does. It is still the case, however, that as the game progresses, S_2 will soon become too small for the results to have much statistical significance and AOSP's play will begin to resemble random play.

- Each board generated by LOS is unlikely to be consistent with the current belief state; thus the values computed by LOS are likely to be close to random.
- At the beginning of the game, HS will behave identically to AOSP. As the game proceeds and the size of S_2 decreases, HS will put more and more randomly generated boards into S_3 , thus making the results more noisy. Thus HS's quality of play is likely to be worse than AOSP's.

Analysis 2: If there are n possible states in the game, then the number of moves at each level cannot continue to grow exponentially, but will eventually flatten out. The game “tree” will be a graph rather than a tree, with n nodes (at most) at each depth, one for each possible state. There will be b edges from each node at depth i to nodes at depth $i + 1$, $1/c$ of which are consistent with any given observation; suppose these edges go to a random set of nodes. Then for each state s , the probability (under certain independence assumptions) that it is reachable in i moves is about $\min(1, (n - 1)^{i-3}((b/c)/(n - 1))^{i-1})$. In other words, the probability that a randomly chosen state s has a history consistent with O_{ij} approaches 1 at exponentially. This suggests the following:

- As the game proceeds, the amount of time needed by AOS will eventually level off; its upper bound will be n times the amount of time needed by each call to the classical game-tree search algorithm.
- Rather than degrading to random play as in Analysis 1, AOSP's quality of play will eventually level off at some level above that, depending on the number of states available in the pool.
- As the game proceeds, the probability of a randomly generated board being consistent with the current belief state will increase toward 1; thus LOS will produce increasingly good quality of play. However, its play will be limited by the fact that it has no good way to assign relative probabilities to its randomly generated boards.
- At the beginning of the game, HS will behave identically to AOSP. As the game proceeds and AOSP's sample size decreases, HS will fill up the rest of the sample with randomly generated boards—but as the game proceeds, it will become increasingly likely that these randomly chosen boards are consistent with the current belief state. Thus HS's quality of play is likely to be better than AOSP's.

Summary. With the first set of assumptions, AOSP is likely to perform much better than LOS, and somewhat better than HS. With the second set of assumptions, it is unclear which of LOS and AOSP will be better, but HS is likely to perform better than AOS, AOSP, and LOS.

Since both sets of assumptions represent extremal cases, our hypotheses are that in many games the actual performance is likely to be somewhere in between, such that HS will perform somewhat better than AOSP. In other words, we hypothesize that in many games, if last-observation consistent boards are included in the statistical sample later in the game, this will help rather than hurt the evaluations. Section 5 describes our experimental test of that hypothesis.

3 Kriegspiel Chess

As our test domain, we chose the game of *kriegspiel chess* [Li, 1994; 1995]. This game is like chess except that neither player can see the other's pieces. When player x captures one of player y 's pieces, a referee announces that he/she has made a capture but not what piece was captured, and the referee removes the captured piece from y 's board but does not say what piece captured it. When x tries to make a move that is illegal (an attempted pawn take, or moving into check, or attempting to jump an opponent's piece), the referee announces that the move is illegal but not why. When x puts y 's king in check, the referee tells both players that y 's king is in check, and gives them partial information about how the check occurred (namely by rank, file, long diagonal, short diagonal, knight, or some combination of the above). Both players hear all referee announcements.

Twenty moves into a kriegspiel chess game, a conservative estimate is that at each node of the game tree, the current sequence of observations is consistent with more than 10^{13} board positions. Because of this uncertainty, kriegspiel chess is a notoriously difficult game to win [Li, 1994; 1995]; most games end in draws.

4 Implementation of the Algorithms

For Choose-move's Evaluate-board subroutine, we took the GPL'ed chess program provided by GNU and modified it to return a minimax value for a particular board. Note that for each $s \in S$ we call the evaluation function $|M|$ times, where $|M|$ is the number of applicable moves (20-40 in kriegspiel). The only exception to this is the case where a move $m \in M$ is not legal in some state $s \in S$; in this case we omit s when computing the average value for m . As shown in Figure 1, we then return the move with the highest average value.

4.1 Algorithms with Time Limits

In order to make fair comparisons among LOS, AOSP, and HS, they cannot be implemented in the exact way described in Section 2. They must be modified so that one of the inputs to the algorithm is t , the amount of time available to decide on a move, so that the algorithm can do as well as it can in that amount of time. The modified algorithms—Timed LOS, Timed AOSP, and Timed HS—are described below.

We have implemented all three of these algorithms, using a combination of C and C++. In the next few months, we intend to make our implementations publicly available, both as open-source software and as a kriegspiel-chess game server running on our web site.

Timed LOS: Rather than taking the set S as input as shown in Figure 1, Timed LOS generates the members of S one at

a time and evaluates them as they are generated, so that it can generate and evaluate as many boards as it can during the time available. Once the time is up, it returns the move whose average value is highest, as shown in Figure 1.

Timed AOSP: Timed AOSP maintains a pool of states $P = \{s_1, \dots, s_p\}$ that are known to be consistent with the current belief state b . Using an estimate of how long Evaluate-board will take on each board, it calculates some number of boards k_t that it can evaluate during the available time t . The estimate is deliberately a little low, to try to keep Timed AOSP from running overtime and to ensure that there will be time left over to attempt to generate more consistent boards. There are three cases:

- If $p \geq k_t$ then Timed AOSP calls **Choose-move**($\{s_1, \dots, s_{k_t}\}$), and returns the recommended move.
- If $0 < p < k_t$ then Timed AOSP calls **Choose-move**(P), and returns the recommended move.
- If $p = 0$ then Timed AOSP returns a random move.

During whatever remains of the available time, AOSP tries to generate more histories that are consistent with b ; and for every such history, it adds the resultant board to the pool (see section 4.2 below).

Each time the referee makes an announcement, Timed AOSP must update the pool to be consistent with the announcement. This can cause the pool to either shrink (when Timed AOSP is told a move is illegal) or to grow (when Timed AOSP is told that the opponent has moved). This computation occurs at the beginning of AOSP's turn.

If the pool were allowed to grow unchecked, it could potentially get quite large; hence we limit its size to 20,000 boards. If the number of boards in the pool ever goes higher than this, we remove enough boards to get to the number of boards down to 10,000. Because the 30 second time limit allows only enough time to call **Choose-move** on a set about 350 boards from the pool, this is believed adequate.

Timed HS: Timed HS works the same as Timed AOSP, with one exception. If $0 \leq p < k_t$, then Timed HS generates a set R of $p - k_t$ random boards that are consistent with o_{ij} (called *last-observation* consistent boards), and calls **Choose-move**($P \cup R$). This rules out the possibility of ever having to make a random move. It also restricts the amount of time that Timed HS can spend generating additional boards to put into the pool.

4.2 Consistent History Generation

The algorithm for generating additional histories consistent with b does a depth first search through the space of game histories. At each node, the known game history (O_{ij}) specifies the possible branches. Whenever the algorithm reaches a node with multiple branches (corresponding to an opponent's hidden move in the game history), it randomly orders the branches and proceeds searching according to that order.

5 Experiments

Our experimental hypotheses (based on the analyses in Section 2) were that (1) Timed LOS would perform better than random play, (2) Timed AOSP would perform better than

Timed LOS, and (3) Timed HS would perform somewhat better than Timed AOSP. The third hypothesis caused some controversy because it was based on a notion that not all of the authors believed: that the computation time spent introducing and evaluating last-observation consistent boards would not be better spent trying to find and evaluate more all-observation consistent boards.

To test our hypotheses, we played all three algorithms against each other and against a player who moved at random. Each player plays approximately half of the games as white and half of the games as black. All experiments were run on Xeon 2.6GHz chips with 500 MB RAM, running Linux. Each player was allowed to spend 30 seconds deciding each move, including moves which are decided after an attempted illegal move.

Figure 2 shows the results for the algorithms versus the random player, Figure 3 shows the results of head-to-head tests of the three algorithms. The results include the wins/losses/draws, and each player's average material value at each move of the game (using the standard chess value for each piece). We observe the following:

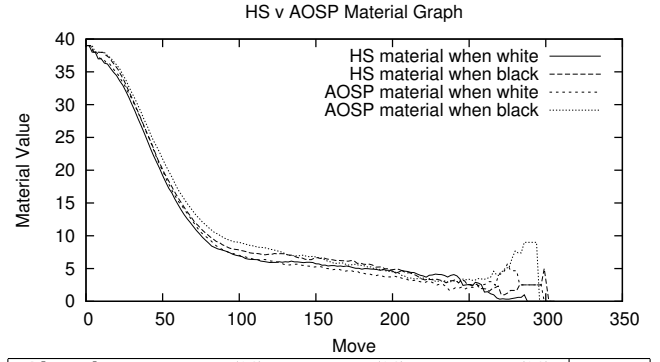
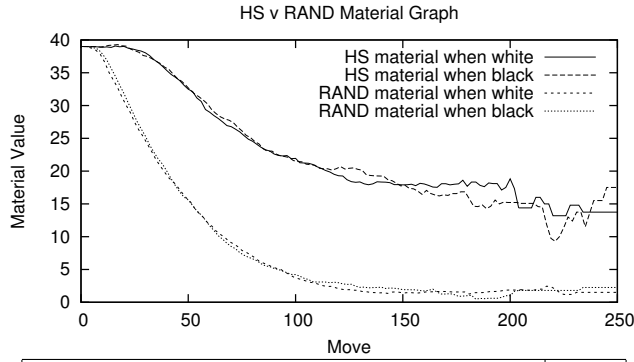
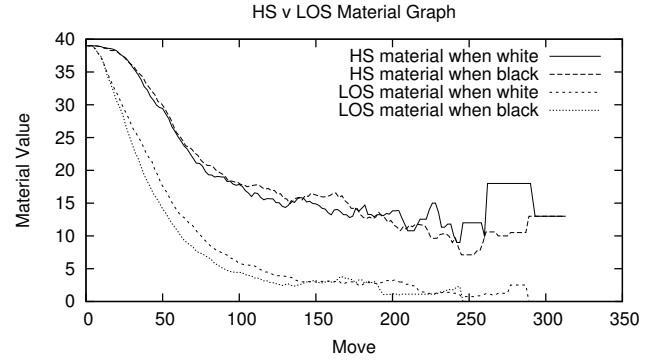
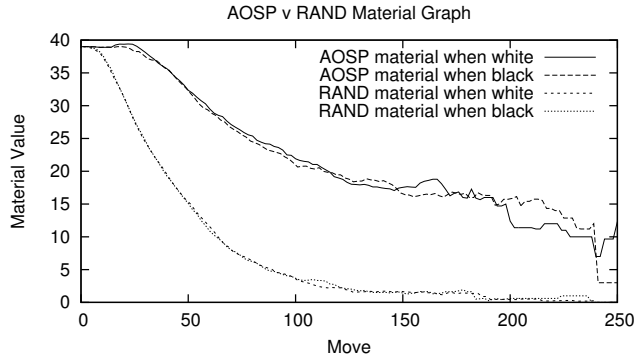
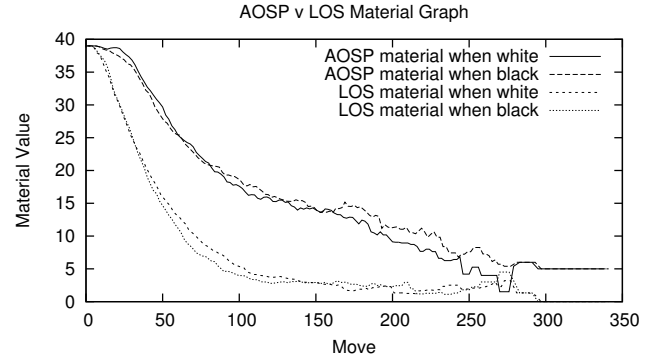
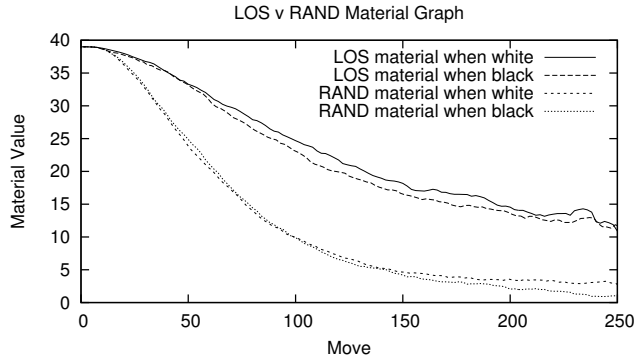
1. All three algorithms played much better than the random player, confirming our first hypothesis. The large number of draws is unsurprising, since kriegspiel chess is a notoriously difficult game to win.
2. Timed AOSP and Timed HS both played much better than Timed LOS, confirming our second hypothesis.
3. Timed AOSP and Timed HS played about equally well versus a random player, but Timed HS did significantly better than Timed AOSP when the two were played head-to-head. Furthermore, as shown in Figure 4, Timed HS did this using a substantial number of last-observation consistent boards as the game progressed.

The third observation is very interesting, because it suggests that our hypothesis about last-observation consistent boards is correct: they become more useful as the game progresses, because they are more likely to be consistent with the current belief state. Even though we do not know what probabilities to assign to them in the last line of **Choose-move**, they still provide useful information.

6 Related Work

The best-known work on imperfect-information games includes work on the games of bridge [Smith *et al.*, 1998; Ginsberg, 1999], scrabble [Sheppard, 2002], and poker [Billings *et al.*, 2003]; we discussed this work in Section 1.

The existing literature on kriegspiel chess is rather small, and we believe ours is the first attempt to create a kriegspiel chess program capable of reasonable play throughout the entire game. Two books have been written on how to play kriegspiel chess [Li, 1994; 1995]. [Wetherell *et al.*, 1972] have implemented a program to act as the referee who sees the entire kriegspiel-chess game board and tells each player the information described in the second paragraph of Section 3. [Sakuta *et al.*, 2001] have implemented a search strategy for some imperfect-information games that are simpler than



Algorithms	Win (%)	Loss (%)	Draw (%)	Runs
LOS v rand	39 ± 2	0 ± 0.3	61 ± 2	559
AOSP v rand	63 ± 2	0 ± 0.3	37 ± 2	560
HS v rand	65 ± 2	0.5 ± 0.3	35 ± 2	558

Algorithms	Win (%)	Loss (%)	Draw (%)	Runs
AOSP v LOS	31 ± 4.8	0 ± 1	69 ± 4.8	190
HS v LOS	38 ± 5	0.5 ± 1	61 ± 5	190
HS v AOSP	13.3 ± 0.4	10.7 ± 0.4	76 ± 0.5	1669

Figure 2: Wins/losses/draws percentages plus or minus a 95% confidence interval and average material value at each move, in games between Timed LOS, Timed AOSP, Timed HS, and a random player. Material value is a heuristic measure of a player's ability. We assign material value as follows: queen is worth 9, rook 5, bishop 3, knight 3, and pawn 1. The sum of all a player's pieces material value after a particular move is the player's material value for that move.

Figure 3: Wins/losses/draws percentages plus or minus a 95% confidence interval and average material value at each move, in head-to-head comparisons of Timed LOS, Timed AOSP, and Timed HS.

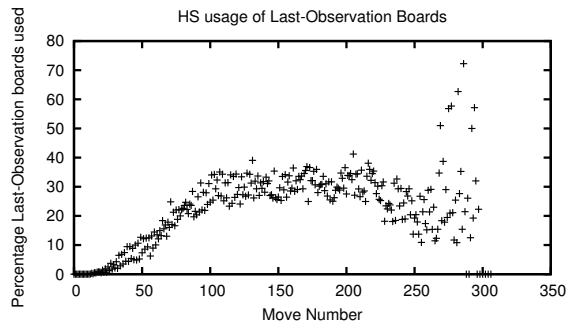


Figure 4: The average percentage of last-observation consistent boards that Timed HS used at each move in its games against Timed AOSP.

kriegspiel, and [Bolognesi and Ciancarini, 2004] and have developed search strategies for kriegspiel-chess endgames.

7 Conclusion

In games such as kriegspiel chess, the belief states are huge compared to those in bridge, scrabble, and poker. Furthermore, there is not a simple uncertainty model from which to generate boards consistent with a belief state. Our results demonstrate that statistical sampling approaches can nevertheless be developed that do well in such games.

Our results also show that to play well, it is not necessary for the random sample to consist only of game boards that satisfy all of a player's observations. In fact, we were able to win 24% more often by starting out with such boards, but gradually switching over (as the game progressed) to boards that merely are consistent with the latest observation.

The reason for our surprising result is that as the game progresses, a board that is consistent with the last move becomes more and more likely to be consistent with the entire set of observations, even if we have no idea what sequence of moves might have actually generated this board.

7.1 Future Work

The biggest limitation of our work is that we have not yet been able to evaluate play against human opponents. To do so will take some effort: we know of no rating system for kriegspiel chess players, hence it is difficult to tell whether a human player is good or bad. To overcome this obstacle, we are currently building a kriegspiel-chess game server which we will run on the web and will make available as open-source software.² The server will include a ranking system so that we may compare our algorithms against humans.

We have yet to attempt any opponent modeling. For example, we intend to extend our algorithms to assign a weight to each game board by estimating how likely the opponent was to make all of the moves leading to that board.

A well-known deficiency of statistical-sampling approaches is that they do not take into account the information-gathering value of illegal moves [Frank and Basin, 1998]. We

²To construct our kriegspiel server, we are using the Generic Game Server that is available as open-source software at <http://external.nj.nec.com/homepages/igord/gsa-ggs.htm>.

have ideas for how to modify our algorithms to overcome this deficiency, and intend to test these ideas in the near future.

It would be worthwhile to examine different search strategies and evaluation functions. We currently rely on gnu chess for both, but it is not entirely clear whether this is the best approach for a game like kriegspiel chess.

Acknowledgments

This work was supported by the following grants, contracts, and awards: ARO grant DAAD190310202, ARL grants DAAD190320026 and DAAL0197K0135, the ARL CTAs on Telecommunications and Advanced Decision Architectures, NSF grants IIS0329851, 0205489 and IIS0412812, UC Berkeley contract number SA451832441 (subcontract from DARPA's REAL program). The opinions expressed in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

References

- [Applegate *et al.*, 1991] David Applegate, Guy Jacobson, and Daniel Sleator. Computer analysis of sprouts. Technical report, Carnegie Mellon University, 1991.
- [Billings *et al.*, 2003] Darse Billings, N. Burch, Aaron Davidson, Robert Holte, Jonathan Schaeffer, T. Schauenberg, and Duane Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In *IJCAI-03*, pages 661–668, 2003.
- [Bolognesi and Ciancarini, 2004] A. Bolognesi and P. Ciancarini. Searching over metapositions in kriegspiel. In *Computer Games 2004*, 2004.
- [Frank and Basin, 1998] Ian Frank and David A. Basin. Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, 100(1-2):87–123, 1998.
- [Ginsberg, 1999] Matthew L. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *IJCAI-99*, pages 584–589, 1999.
- [Li, 1994] David Li. *Kriegspiel: Chess Under Uncertainty*. Premier, 1994.
- [Li, 1995] David Li. *Chess Detective: Kriegspiel Strategies, Endgames and Problems*. Premier, 1995.
- [Sakuta *et al.*, 2001] M. Sakuta, J. Yoshimura, and H. Iida. A deterministic approach for solving kriegspiel-like problems. In *MSO Computer Olympias Workshop*, 2001.
- [Sheppard, 2002] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.
- [Smith *et al.*, 1998] Stephen J. J. Smith, Dana S. Nau, and Thomas Throop. Computer bridge: A big win for AI planning. *AI Magazine*, 19(2):93–105, 1998.
- [Wetherell *et al.*, 1972] C. S. Wetherell, T. J. Buckholtz, and K. S. Booth. A director for kriegspiel, a variant of chess. *Comput. J.*, 15(1):66–70, 1972.

Counting Solutions of CSPs: A Structural Approach*

Gilles Pesant

ILOG, 1681 route des Dolines, 06560 Valbonne, France
pesant@crt.umontreal.ca

Abstract

Determining the number of solutions of a CSP has several applications in AI, in statistical physics, and in guiding backtrack search heuristics. It is a $\#P$ -complete problem for which some exact and approximate algorithms have been designed. Successful CSP models often use high-arity, global constraints to capture the structure of a problem. This paper exploits such structure and derives polytime evaluations of the number of solutions of individual constraints. These may be combined to approximate the total number of solutions or used to guide search heuristics. We give algorithms for several of the main families of constraints and discuss the possible uses of such solution counts.

1 Introduction

Many important combinatorial problems in Artificial Intelligence (AI), Operations Research and other disciplines can be cast as Constraint Satisfaction Problems (CSP). One is usually interested in finding a solution to a CSP if it exists (the *combinatorial existence problem*) and much scientific literature has been devoted to this subject in the past few decades. Another important question, though not as thoroughly studied, is how many solutions there are (the *combinatorial enumeration problem*). Our ability to answer this question has several applications in AI (e.g. [Orponen, 1990],[Roth, 1996],[Darwiche, 2001]), in statistical physics (e.g. [Burton and Steif, 1994], [Lebowitz and Gallavotti, 1971]), or more recently in guiding backtrack search heuristics to find solutions to CSPs [Horsch and Havens, 2000][Kask *et al.*, 2004][Refalo, 2004]. In this latter context, an estimation of the number of solutions is sought repeatedly as search progresses. The best strategy to keep the overall runtime low may be to trade accuracy for speed while computing these approximations.

The model of a CSP is centered on the constraints, which give it its structure. This paper proposes to exploit this inherent structure to derive polytime approximations to the solution count of a CSP by combining solution counts of component constraints. At the level of the individual constraint,

consistency algorithms act on the domains of the variables in its scope (i.e. on which the constraint is defined) to filter out values which do not belong to any solution of this constraint, thus avoiding some useless search. For many families of constraints, all such values can be removed in polynomial time even though globally the problem is $\mathcal{NP}$ -hard. The only visible effect of the consistency algorithms is on the domains, projecting the set of solutions on each of the variables. But a constraint's consistency algorithm often maintains information which may be exploited to evaluate the number of valid tuples. We intend to show that a little additional work is often sufficient to provide close or even exact solution counts for a constraint, given the existing consistency algorithm.

The global counting problem for CSPs (denoted $\#CSP$) is $\#P$ -complete even if we restrict ourselves to binary constraints and is thus very likely intractable. Recent theoretical work [Bulatov and Dalmau, 2003] characterizes some tractable classes of CSPs for this problem but these are quite restrictive, especially for practical problems. The counting problem for Boolean CSPs ($\#SAT$) is the best studied subclass (see e.g. [Birnbbaum and Lozinskii, 1999]).

For binary CSPs, exponential time exact algorithms have been described (e.g. [Angelsmark and Jonsson, 2003]). For backtrack search, [Kask *et al.*, 2004] adapts Iterative Join-Graph Propagation to approximate the number of solutions extending a partial solution and uses its results in a value-ordering heuristic to solve CSPs, choosing the value whose assignment to the current variable gives the largest approximate solution count. An implementation optimized for binary constraints performs well compared to other popular strategies. [Refalo, 2004] proposes a generic variable-ordering heuristic based on the impact the assignment of a variable has on the reduction of the remaining search space, computed as the Cartesian product of the domains of the variables. It reports promising results on multi-knapsack, magic square, and Latin square completion benchmarks for (not necessarily binary) CSPs. The value- and variable-ordering heuristics described above rely on an approximation of the solution count. Both can benefit from an improvement in the quality and/or efficient computation of this approximation.

In the rest of the paper, Section 2 revisits several families of constraints to evaluate the effort necessary to provide solution counts, Section 3 explores possible uses for them, Section 4 gives some examples, and Section 5 discusses future work.

*Research conducted while the author was on sabbatical leave from École Polytechnique de Montréal.

2 Looking at Constraints

This section proceeds through a short list of usual constraints (for their origin, see e.g. [Régim, 2004]) and examines how solution counts may be derived. We assume throughout that the constraints are domain consistent, unless stated otherwise. Given a variable x , we denote its domain of possible values as $D(x)$, the smallest and largest value in $D(x)$ as $x_{\min}$ and $x_{\max}$ respectively, and the number of values in $D(x)$ as d_x . We use $\#\gamma$ to mean the number of solutions for constraint γ .

2.1 Binary Constraints

Many arc consistency algorithms have been developed for binary constraints of the form $\gamma(x, y)$. Among them, AC-4 [Mohr and Henderson, 1986] maintains a support counter for each value in $D(x)$ and $D(y)$. The total solution count is then simply the sum of those counters over the domain of one of the variables:

$$\#\gamma(x, y) = \sum_{v \in D(x)} sc(x, v) \quad (1)$$

where $sc(x, v)$ stands for the support counter of value v in the domain of variable x . However, other consistency algorithms only compute support witnesses as needed and hence would not have support counters available.

2.2 Arithmetic Binary Constraints

Typically, support counters are not maintained by the consistency algorithm for arithmetic binary constraints but one rather relies on the semantics of the constraint. Take for example $x < y$: knowing $x_{\min}$ and $y_{\max}$ is sufficient to remove all the inconsistent values. To compute the number of valid couples we can simply enumerate them, which done naively would require $\Omega(d_x \cdot d_y)$ time. Provided the domains are maintained in sorted order, a reasonable assumption, that enumeration can be done in $\Theta(d_x + d_y)$ time by using a running pointer in each domain as described in Algorithm 1.

```

function lt_card( $x, y$ )
let  $t_x$  and  $t_y$  be tables containing the values of the
domains of  $x$  and  $y$ , respectively, in increasing order;
 $p := 1$ ;  $q := 1$ ;  $c := 0$ ;
while  $q \leq d_y$  do
  while  $p \leq d_x$  and  $t_x[p] < t_y[q]$  do
     $p := p + 1$ ;
     $c := c + p - 1$ ;
   $q := q + 1$ ;
return  $c$ ;

```

Algorithm 1: Computing the number of solutions to $x < y$.

$$\#(x < y) = \text{lt_card}(x, y) \quad (2)$$

For each value v in $D(y)$, it finds out of the number of supports $sc(y, v)$ from $D(x)$ and adds them up in c . We can make the algorithm incremental by storing $sc(y, v)$ for each v in $D(y)$ and $q(w)$, the smallest q such that $w < t_y[q]$, for each w in $D(x)$: if v is removed from $D(y)$, the total number of solutions decreases by $sc(y, v)$; if w is removed from $D(x)$,

the total number of solutions decreases by $d_y - q(w) + 1$ and $sc(y, v)$ is decremented by one for $q(w) \leq v \leq d_y$.

Sometimes the domains of variables involved in such constraints are maintained as intervals. This simplifies the computation, for which we derive a closed form:

$$\begin{aligned} \#(x < y) &= \sum_{i=y_{\min}-x_{\min}}^{y_{\max}-x_{\min}} i - \sum_{i=1}^{y_{\max}-x_{\max}-1} i \\ &= y_{\max} \cdot (x_{\max} + 1) - x_{\min} \cdot d_y \\ &\quad - \frac{1}{2} \cdot (x_{\max}^2 + x_{\max} + y_{\min}^2 - y_{\min}) \end{aligned} \quad (3)$$

Clearly, what preceded equally applies to constraints $x \leq y$, $x > y$, and $x \geq y$, with small adjustments. Counting solutions for equality and disequality constraints is comparatively straightforward (remember that domain consistency has been achieved):

$$\#(x = y) = d_x \quad (4)$$

$$\#(x \neq y) = d_x \cdot d_y - |D(x) \cap D(y)| \quad (5)$$

With an appropriate representation of the intersection of $D(x)$ and $D(y)$, an incremental version of the count for $x \neq y$ requires constant time.

In some applications such as temporal reasoning, constants are present in the binary constraints (e.g. $x < y + t$ for some constant t). The previous algorithms and formulas are easily adapted to handle such constraints.

2.3 Linear Constraints

Many constraint models feature linear constraints of the form $\sum_{i=1}^n a_i x_i \circ b$ where $\circ$ stands for one of the usual relational operators and the a_i 's and b are integers. Bound consistency is typically enforced, which is a fast but rather weak form of reasoning for these constraints. Working from domains filtered in this way is likely to yield very poor approximations of the number of solutions.

Example 1 Constraint $3x_1 + 2x_2 - x_3 - 2x_4 = 0$ with bound consistent domains $D(x_i) = \{0, 1, 2\}$, $1 \leq i \leq 4$ admits only seven solutions even though the size of the Cartesian product is 81. Since the value of any one variable is totally determined by the combination of values taken by the others, the upper bound can be lowered to 27 but this is still high.

These constraints are actually well studied under the name of linear Diophantine equations and inequations. In particular, [Ajili and Contejean, 1995] offers an algorithm adapted to constraint programming that produces a finite description of the set of solutions of a system of linear Diophantine equations and inequations over $\mathbb{N}$.

When the constraints are of the form $b' \leq \sum_{i=1}^n a_i x_i \leq b$ with the a_i 's and x_i 's belonging to $\mathbb{N}$, we get knapsack constraints for which [Trick, 2003] adapts a pseudo-polynomial dynamic programming algorithm. It builds an acyclic graph $G_{\langle a_i \rangle, \langle x_i \rangle, b, b'}$ of size $\mathcal{O}(nb)$ in time $\mathcal{O}(nbd)$ (where $d = \max\{d_{x_i}\}$) whose paths from the initial node to a goal node correspond to solutions. It is pointed out that enumerating solutions amounts to enumerating paths through a depth-first search, in time linear in the size of the graph.

$$\#(b' \leq \sum_{i=1}^n a_i x_i \leq b) = |\{\text{paths traversing } G_{\langle a_i \rangle, \langle x_i \rangle, b, b'}\}| \quad (6)$$

Interestingly, the approach could be generalized so as to lift the nonnegativity restriction on the coefficients and variables at the expense of a larger graph of size $\mathcal{O}(nr)$ where r represents the magnitude of the range of $\sum_{i=1}^n a_i x_i$ over the finite domains.

2.4 Element Constraints

The ability to index an array with a finite-domain variable, commonly known as an **element** constraint, is often present in practical models. Given variables x and i and array of values a , **element**(i, a, x) constrains x to be equal to the i^{th} element of a ($x = a[i]$). Note that this is not an arbitrary relation between two variables but rather a functional relationship. As a relation, it could potentially number $d_x \cdot d_i$ solutions, corresponding to the Cartesian product of the variables. As a functional relationship, there are exactly as many tuples as there are consistent values for i :

$$\# \text{element}(i, a, x) = d_i \quad (7)$$

This exact count improves on the product of the domain sizes by a factor d_x .

2.5 Regular Language Membership Constraints

Given a sequence of variables $X = \langle x_1, x_2, \dots, x_n \rangle$ and a deterministic finite automaton $\mathcal{A}$, **regular**($X, \mathcal{A}$) constrains any sequence of values taken by the variables of X to belong to the regular language recognized by $\mathcal{A}$ [Pesant, 2004]. The consistency algorithm for this constraint builds a directed acyclic graph $G_{X, \mathcal{A}}$ whose structure is very similar to the one used by [Trick, 2003] for knapsack constraints, as reported in Section 2.3. Here as well, each path corresponds to a solution of the constraint and counting them represents no complexity overhead with respect to the consistency algorithm.

$$\# \text{regular}(X, \mathcal{A}) = |\{\text{paths traversing } G_{X, \mathcal{A}}\}| \quad (8)$$

2.6 Among Constraints

Under constraint **among**(c, X, V), variable c corresponds to the number of variables from set X taking their value from set V [Beldiceanu and Contejean, 1994]. Let $R = \{x \in X : D(x) \subseteq V\}$, the subset of variables required to take a value from V , and $P = \{x \in X \setminus R : D(x) \cap V \neq \emptyset\}$, the subset of variables possibly taking a value from V . Those two sets are usually maintained by the consistency algorithm since the relationship $|R| \leq c \leq |R| + |P|$ is useful to filter the domain of c .

Define ν_S as the number of variables from set S taking their value from V . To derive the number of solutions, we first make the following observations:

1. $\nu_R = |R|$, a constant, so that in any solution to the constraint the value taken by $x \in R$ can be replaced by any other in its domain and it remains a solution.
2. $\nu_{X \setminus (R \cup P)} = 0$, a constant, so that in any solution to the constraint the value taken by $x \in X \setminus (R \cup P)$ can be replaced by any other in its domain and it remains a solution.

3. Any assignment of the variables of P such that $\nu_P = c - |R|$ of them take a value in V can be extended into $\prod_{x \in X \setminus P} d_x$ complete solutions.
4. There are such assignments for every value of ν_P from $c_{\min} - |R|$ to $c_{\max} - |R|$ and for every subset S of P identifying the ν_P variables in question. How many assignments, given S ? Each variable $y \in S$ has $|D(y) \cap V|$ possible values and each variable $z \in P \setminus S$ has $|D(z) \setminus V|$ possible values, totaling $\#a(S) = \prod_{y \in S} |D(y) \cap V| \cdot \prod_{z \in P \setminus S} |D(z) \setminus V|$ assignments.

Putting it all together gives:

$$\# \text{among}(c, X, V) = \prod_{x \in X \setminus P} d_x \cdot \sum_{k=c_{\min}}^{c_{\max}} \sum_{\substack{S \subseteq P \\ |S|=k-|R|}} \#a(S) \quad (9)$$

In the worst case, the number of S sets to consider is exponential in the number of variables. It may be preferable sometimes to compute faster approximations. We can replace every $\#a(S)$ by some constant lower or upper bound. Let t_{in} be a table in which the sizes of $D(x) \cap V$ for every $x \in P$ appear in increasing order. Similarly, let t_{out} be a table in which the sizes of $D(x) \setminus V$ for every $x \in P$ appear in increasing order. We define a lower bound $\ell(a(k)) = \prod_{i=1}^{k-|R|} t_{\text{in}}[i] \cdot \prod_{i=1}^{|P|-k+|R|} t_{\text{out}}[i]$ and an upper bound $u(a(k)) = \prod_{i=1}^{k-|R|} t_{\text{in}}[|P|-i+1] \cdot \prod_{i=1}^{|P|-k+|R|} t_{\text{out}}[|P|-i+1]$. Then

$$\# \text{among}(c, X, V) \geq \prod_{x \in X \setminus P} d_x \cdot \sum_{k=c_{\min}}^{c_{\max}} \binom{|P|}{k-|R|} \cdot \ell(a(k)) \quad (10)$$

$$\# \text{among}(c, X, V) \leq \prod_{x \in X \setminus P} d_x \cdot \sum_{k=c_{\min}}^{c_{\max}} \binom{|P|}{k-|R|} \cdot u(a(k)) \quad (11)$$

Note that if the variables in P have identical domains then the lower and upper bounds coincide and we obtain the exact count.

Example 2 Consider **among**($c, \{x_1, x_2, x_3, x_4, x_5\}, \{1, 2\}$) with $D(c) = \{3, 4\}$, $D(x_1) = \{1, 3, 4\}$, $D(x_2) = \{1, 2\}$, $D(x_3) = \{3, 4\}$, $D(x_4) = \{2\}$, and $D(x_5) = \{1, 2, 3\}$. We obtain $R = \{x_2, x_4\}$, $P = \{x_1, x_3, x_5\}$, and $\# \text{among}(c, \{x_1, x_2, x_3, x_4, x_5\}, \{1, 2\}) = d_{x_2} \cdot d_{x_4} \cdot (|D(x_1) \cap \{1, 2\}| \cdot |D(x_3) \setminus \{1, 2\}| + |D(x_3) \cap \{1, 2\}| \cdot |D(x_1) \setminus \{1, 2\}| + |D(x_1) \cap \{1, 2\}| \cdot |D(x_5) \cap \{1, 2\}|) = 2 \cdot 2 \cdot 1 \cdot (1 \cdot 1 + 2 \cdot 2 + 1 \cdot 2) = 28$, by (9). Using (10), (11) we get $16 \leq \# \text{among}(c, \{x_1, x_2, x_3, x_4, x_5\}, \{1, 2\}) \leq 40$. In comparison, the size of the Cartesian product is 72.

2.7 Mutual Exclusion and Global Cardinality Constraints

Besides **among**, other constraints are concerned with the number of repetitions of values. Constraint **alldiff**(X) forces the variables of set X to take different values. Constraint **gcc**(Y, V, X), for a sequence of variables $Y = \langle y_1, y_2, \dots, y_m \rangle$ and a sequence of values $V = \langle v_1, v_2, \dots, v_m \rangle$, makes each y_i equal to the number of times

a variable of X takes value v_i . It is a generalization of the former.

Already for the `alldiff` constraint, the counting problem includes as a special case the number of perfect matchings in a bipartite graph, itself equivalent to the $\mathcal{NP}$ -hard problem of computing the permanent of the adjacency matrix representation of the graph. So currently we do not know of an efficient way to compute $\#alldiff(X)$ or $\#gcc(Y, V, X)$ exactly. A polytime randomized approximation algorithm for the permanent was recently proposed in [Jerrum *et al.*, 2004] but its time complexity, in $\Omega(n^{10})$, remains prohibitive. Therefore even getting a reasonable approximate figure is challenging. We nevertheless propose some bounds.

Upper Bound

If $D(x) \subseteq D(y)$ for some variables $x, y \in X$ then regardless of the value v taken by x we know that at most $d_y - 1$ possibilities remain for y since $v \in D(y)$. We generalize this simple observation by considering $\mathcal{D} = \{D(x) : x \in X\}$, the distinct domains of X . For each $D \in \mathcal{D}$, define $E_D = \{x \in X : D(x) = D\}$, the set of variables with domain D , and $S_D = \{x \in X : D(x) \subset D\}$, the set of variables whose domain is properly contained in D . Then

$$\#alldiff(X) \leq \prod_{D \in \mathcal{D}} \prod_{i=1}^{|E_D|} (|D| - |S_D| - i + 1) \quad (12)$$

Computing it requires taking the product of n terms, each of which can be computed easily in $\mathcal{O}(nm)$ time and probably much faster with an appropriate data structure maintaining the E_D and S_D sets. This upper bound is interesting because for the special case where all domains are equal, say $\{v_1, v_2, \dots, v_m\}$, it simplifies to the exact solution count,

$$\prod_{i=1}^n (m - i + 1) = m! / (m - n)!$$

Lower Bound

As hinted before, the consistency algorithm for `alldiff` computes a maximum matching in a bipartite graph. It is known that every maximum matching of a graph can be obtained from a given maximum matching by successive transformations through even-length alternating paths starting at an unmatched vertex or through alternating cycles.¹ Each transformation reverses the status of the edges on the path or cycle in question: if it belonged to the matching it no longer does, and vice versa. The result is necessarily a matching of the same size. Finding every maximum matching in this way (or any other) would be too costly, as we established before, but finding those that are just one transformation away is fast and provides a lower bound on the number of maximum matchings, or equivalently on the solution count of `alldiff`.

By not applying transformations in succession, we also avoid having to check whether two matchings obtained by distinct sequences of transformations are in fact identical,

¹An *alternating* path or cycle alternates between edges belonging to a given matching and those not belonging to it.

which could happen. However, successive transformations from distinct connected components is safe: each resulting matching is distinct. We may therefore take the product of the number of maximum (sub)matchings in each connected component of the bipartite graph. Identifying the connected components C takes time linear in the size of the graph. Enumerating all the appropriate cycles and paths of each connected component can be done in $\mathcal{O}(nm^2)$ time. Let their number be ν_g for each connected component $g \in C$. Then

$$\#alldiff(X) \geq \prod_{g \in C} (\nu_g + 1) \quad (13)$$

For the `gcc` constraint, we may be able to adapt the ideas which gave rise to the previous lower bound. The consistency algorithm for this constraint is usually based on network flows. Given a network and a flow through it, a residual graph can be defined and the circuits in this graph lead to equivalent flows. Another idea is to decompose the constraint into among constraints on singleton sets of values and use (9)-(11).

3 Using Solution Counts of Constraints

This section examines the possible uses of solution counts of constraints.

3.1 Approximate the Solution Count of a CSP

They can be used to approximate the number of solutions to a CSP. Given a model for a CSP, consider its variables as a set S and its constraints as a collection C of subsets of S defined by the scope of each constraint. Add to C a singleton for every variable in S . A *set partition* of S is a subset of C whose elements are pairwise disjoint and whose union equals S (see Section 4 for some examples). To each element of the partition corresponds a constraint (or a variable in case of a singleton) for which an upper bound on the number of solutions can be computed (or taken as the cardinality of the domain in case of a singleton). The product of these upper bounds gives an upper bound on the total number of solutions. In general there are several such partitions possible and we can find the smallest product over them. If it is used in the course of a computation as variables become fixed and can be excluded from the partition, new possible partitions emerge and may improve the solution count.

3.2 Guide Search Heuristics

As we saw in the introduction, these counts are also useful to develop robust search heuristics. They may allow a finer evaluation of impact in the generic search heuristic of [Re-falo, 2004] since their approximation of the size of the search space is no worse than the Cartesian product of the domains. Value-ordering heuristics such as [Kask *et al.*, 2004] are also good candidates. Other search heuristics following the *first-fail principle* (detect failure as early as possible) and centered on constraints can be guided by a count of the number of solutions left for each constraint. We might focus the search on the constraint currently having the smallest number of solutions, recognizing that failure necessarily occurs through a constraint admitting no more solutions (see Section 4).

3.3 Evaluate Projection Tightness

We can also compute the ratio of the (approximate) solution count of a constraint to the size of the Cartesian product of the appropriate domains, in a way measuring the tightness of the projection of the constraint onto the individual variables. A low ratio stresses the poor quality of the information propagated to the other constraints (i.e. the filtered domains) and identifies constraints whose consistency algorithm may be improved. Tightness can also serve as another search heuristic, focusing on the constraint currently exhibiting the worst tightness (i.e. the smallest ratio). An ideal ratio of one corresponds to a constraint perfectly captured by the current domains of its variables.

4 Some Examples

The purpose of this section is to illustrate solution counts and their possible uses through a few simple models, easier to analyse.

4.1 Map Coloring

Consider the well-known map coloring problem for six European countries: Belgium, Denmark, France, Germany, Luxembourg, and the Netherlands. Any two countries sharing a border cannot be of the same color. To each country we associate a variable (named after its first letter) that represents its color. Constraints

$$f \neq b, f \neq \ell, f \neq g, \ell \neq g, \ell \neq b, b \neq n, g \neq n, g \neq d, g \neq b$$

model the problem. If we allow five colors, there are 1440 legal colorings. The cardinality of the Cartesian product of the domains of the variables initially overestimates that number to be 15625 (5^6). A set partition such as $\{f \neq \ell, b \neq n, g \neq d\}$ provides a slightly better estimate, using (5):

$$(d_x \cdot d_y - |D(x) \cap D(y)|)^3 = (5 \cdot 5 - 5)^3 = 8000.$$

Noticing that four of these countries are all pairwise adjacent, an alternate model is

$$\text{alldiff}(\{b, f, g, \ell\}), b \neq n, g \neq n, g \neq d$$

yielding a still better upper bound on the solution count, $5! \cdot 5^2 = 3000$, using (12) with set partition $\{\text{alldiff}(\{b, f, g, \ell\}), n, d\}$. We see here that the solution counts of the constraints in a CSP model, particularly those of high arity, can quickly provide good approximations of the number of solutions of CSPs. The projection tightness of $b \neq n, g \neq n$, and $g \neq d$ is $20/25 = 0.8$; that of $\text{alldiff}(\{b, f, g, \ell\})$ is $5!/5^4 = 0.192$. This could be taken as an indication that search should focus on the latter.

Let us now analyse the effect of coloring one of the countries red. Regardless of the country we choose, the number of legal colorings will decrease five fold to 288 since colors are interchangeable. The true impact of fixing a variable, measured as 1 minus the ratio of the remaining search space after and before ([Refalo, 2004]), is thus 0.8. Table 4.1 reports after each country is colored red the cardinality of the Cartesian product of the domains (column 2), the upper bound on the solution count from each of the two set partitions (columns

var	domains		1st partition		2nd partition	
	solns	impact	solns	impact	solns	impact
ℓ	1600	.898	1024	.872	600	.800
f	1600	.898	1024	.872	600	.800
b	1280	.918	768	.904	480	.840
g	1024	.934	576	.928	384	.872
n	2000	.872	1280	.840	360	.880
d	2500	.840	1600	.800	480	.840
avg	—	.893	—	.869	—	.839

Table 1: Comparing different approximations of the number of solutions of a small map coloring problem.

4 and 6), and their corresponding impacts (columns 3, 5, and 7).

Again we note a significant improvement in the approximation of the number of solutions over the Cartesian product, particularly from the second set partition based on the model using an `alldiff` constraint. The latter also provides a closer approximation of the true impact: the average computed impact is less than 5% away whereas it is more than 11% away for the average computed impact using Cartesian products.

4.2 Rostering

Consider next a simple rostering problem: a number of daily jobs must be carried out by employees while respecting some labor regulations. Let E be the set of employees, D the days of the planning horizon, and J the set of possible jobs, including a day off. Define variables $j_{de} \in J$ ($d \in D, e \in E$) to represent the job carried out by employee e on day d . To ensure that every job is performed on any given day, we can state the following constraints:

$$\text{gcc}(\langle 1, 1, \dots, 1 \rangle, J \setminus \{\text{day off}\}, (j_{de})_{e \in E}) \quad d \in D. \quad (14)$$

Some jobs are considered more demanding than others and we wish to balance the workload among employees. We introduce variables w_{de} representing the workload of employee e on day d and link them to the main variables in the following way:

$$\text{element}(j_{de}, (\ell_j)_{j \in J}, w_{de}) \quad d \in D, e \in E \quad (15)$$

where ℓ_j corresponds to the load of job j . We then add constraints enforcing a maximum workload k :

$$\sum_{d \in D} w_{de} \leq k \quad e \in E. \quad (16)$$

Finally work patterns may be imposed on individual rosters:

$$\text{regular}((j_{de})_{d \in D}, \mathcal{A}) \quad e \in E \quad (17)$$

with the appropriate automaton $\mathcal{A}$.

To approximate the number of solutions, there are a few possibilities for a set partition of the variables $(j_{de})_{d \in D, e \in E}$ and $(w_{de})_{d \in D, e \in E}$: constraints (14)(16) are one, constraints (17)(16) are another, and so are constraints (15). In deciding which one to use, its size (the number of parts), the quality of the bounds, and the projection tightness of each constraint may help. Partition (15) has size $|D| \cdot |E|$ compared to $|D| + |E|$ and $2 \cdot |E|$ for the other two: a smaller size

means larger parts and probably a better overall approximation since each solution count has a more global view. We presented exact counts for every constraint used here except (14), making partition (14)(16) less attractive. A set partition that includes constraints with low tightness (which is difficult to assess here without precise numbers) is more likely to do significantly better than the size of the Cartesian product of the individual domains, which can be considered a baseline.

5 Discussion

This paper argued that looking at the number of solutions of individual constraints in CSPs is interesting and useful. It can approximate the number of solutions as a whole or help guide search heuristics. Efficient ways of counting solutions were given for several families of constraints. The concepts of set partition over the variables and of projection tightness were defined in order to combine solution counts and measure the efficacy of consistency algorithms, respectively.

But this is a first step and many questions remain. Some of the main families of constraints have been investigated but others were left out: for example, those useful in scheduling or packing problems such as `cumulative` and `diffn`, or those for routing problems such as `cycle`. How close can we get to the solution count for them and at what computational cost? For the simplest constraints that we investigated, we mentioned how the algorithms computing solution counts could be made incremental. This is an important issue to achieve efficiency when such counts are computed repeatedly as in backtrack search. What can be done for the other constraints? The whole question of the usefulness of such an approach hasn't been settled either. The answer is likely to come, at least in part, from empirical evidence. Computational experiments will need to be run on larger and more realistic models.

Acknowledgements

Philippe Refalo's work on impact-based search sparked the idea for this paper. I thank Jean-Charles Régin for discussions and the anonymous referees for their constructive comments. This work was partially supported by the Canadian Natural Sciences and Engineering Research Council under grant OGP0218028.

References

- [Ajili and Contejean, 1995] F. Ajili and E. Contejean. Complete Solving of Linear Diophantine Equations and Inequations without Adding Variables. In *Proc. CP'05*, pages 1–17. Springer-Verlag LNCS 976, 1995.
- [Angelsmark and Jonsson, 2003] O. Angelsmark and P. Jonsson. Improved Algorithms for Counting Solutions in Constraint Satisfaction Problems. In *Proc. CP'03*, pages 81–95. Springer-Verlag LNCS 2833, 2003.
- [Beldiceanu and Contejean, 1994] N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathematical and Computer Modelling*, 20:97–123, 1994.
- [Birnbbaum and Lozinskii, 1999] E. Birnbbaum and E. L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *Journal of Artificial Intelligence Research*, 10:457–477, 1999.
- [Bulatov and Dalmau, 2003] A.A. Bulatov and V. Dalmau. Towards a Dichotomy Theorem for the Counting Constraint Satisfaction Problem. In *Proc. FOCS'03*, pages 562–573. IEEE Computer Society, 2003.
- [Burton and Steif, 1994] R. Burton and J. Steif. Nonuniqueness of Measures of Maximal Entropy for Subshifts of Finite Type. *Ergodic Theory and Dynamical Systems*, 14:213–236, 1994.
- [Darwiche, 2001] A. Darwiche. On the Tractable Counting of Theory Models and its Applications to Truth Maintenance and Belief Revision. *Journal of Applied Non-Classical Logic*, 11:11–34, 2001.
- [Horsch and Havens, 2000] M. Horsch and B. Havens. Probabilistic Arc Consistency: A Connection Between Constraint Reasoning and Probabilistic Reasoning. In *UAI-2000*, pages 282–290, 2000.
- [Jerrum et al., 2004] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Non-Negative Entries. *Journal of the ACM*, 51:671–697, 2004.
- [Kask et al., 2004] K. Kask, R. Dechter, and V. Gogate. Counting-Based Look-Ahead Schemes for Constraint Satisfaction. In *Proc. CP'04*, pages 317–331. Springer-Verlag LNCS 3258, 2004.
- [Lebowitz and Gallavotti, 1971] J. Lebowitz and G. Gallavotti. Phase Transitions in Binary Lattice Gases. *Journal of Mathematical Physics*, 12:1129–1133, 1971.
- [Mohr and Henderson, 1986] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.
- [Orponen, 1990] P. Orponen. Dempster's Rule of Combination is #-Complete. *Artificial Intelligence*, 44:245–253, 1990.
- [Pesant, 2004] G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *Proc. CP'04*, pages 482–495. Springer-Verlag LNCS 3258, 2004.
- [Refalo, 2004] P. Refalo. Impact-Based Search Strategies for Constraint Programming. In *Proc. CP'04*, pages 557–571. Springer-Verlag LNCS 3258, 2004.
- [Régin, 2004] J.-C. Régin. Global Constraints and Filtering Algorithms. In M. Milano, editor, *Constraint and Integer Programming: Toward a Unified Methodology*. Kluwer, 2004.
- [Roth, 1996] D. Roth. On the Hardness of Approximate Reasoning. *Artificial Intelligence*, 82:273–302, 1996.
- [Trick, 2003] M.A. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research*, 118:73–84, 2003.

A Scalable Method for Multiagent Constraint Optimization

Adrian Petcu and Boi Faltings

{adrian.petcu, boi.faltings}@epfl.ch

<http://liawww.epfl.ch/>

Artificial Intelligence Laboratory

Ecole Polytechnique Fédérale de Lausanne (EPFL)

IN (Ecublens), CH-1015 Lausanne, Switzerland

Abstract

We present in this paper a new, complete method for distributed constraint optimization, based on *dynamic programming*. It is a utility propagation method, inspired by the *sum-product algorithm*, which is correct only for tree-shaped constraint networks. In this paper, we show how to extend that algorithm to arbitrary topologies using a pseudotree arrangement of the problem graph. Our algorithm requires a *linear number of messages*, whose maximal size depends on the *induced width* along the particular pseudotree chosen.

We compare our algorithm with backtracking algorithms, and present experimental results. For some problem types we report orders of magnitude fewer messages, and the ability to deal with arbitrarily large problems. Our algorithm is formulated for optimization problems, but can be easily applied to satisfaction problems as well.

1 Introduction

Distributed Constraint Satisfaction (DisCSP) was first studied by Yokoo [Yokoo *et al.*, 1992] and has recently attracted increasing interest. In distributed constraint satisfaction each variable and constraint is owned by an agent. Systematic search algorithms for solving DisCSP are generally derived from depth-first search algorithms based on some form of backtracking [Silaghi *et al.*, 2000; Yokoo *et al.*, 1998; Meisels and Zivan, 2003; Hamadi *et al.*, 1998].

Recently, the paradigm of asynchronous distributed search has been extended to constraint optimization by integrating a bound propagation mechanism (ADOPT - [Modi *et al.*, 2003]).

In general, optimization problems are much harder to solve than DisCSP ones, as the goal is not just to find *any* solution, but the *best* one, thus requiring more exploration of the search space. The common goal of all distributed algorithms is to minimize the number of messages required to find a solution.

Backtracking algorithms are very popular in centralized systems because they require very little memory. In a distributed implementation, however, they may not be the best basis since in backtrack search, control shifts rapidly between

different variables. Every state change in a distributed backtrack algorithm requires at least one message; in the worst case, even in a parallel algorithm there will be exponentially many state changes [Kasif, 1986], thus resulting in exponentially many messages. So far, this has been a serious drawback for the application of distributed algorithms in the real world, especially for optimization problems (also noted in [Maheswaran *et al.*, 2004]).

This leads us to believe that other search paradigms, in particular those based on dynamic programming, may be more appropriate for DisCSP. For example, an algorithm that incrementally computes the set of all partial solutions for all previous variables according to a certain order would only use a linear number of messages. However, the messages could grow exponentially in size, and the algorithm would not have any parallelism.

Recently, the sum-product algorithm [Kschischang *et al.*, 2001] has been proposed for certain constraint satisfaction problems, for example decoding. It is an acceptable compromise as it combines a dynamic-programming style exploration of a search space with a fixed message size, and can easily be implemented in a distributed fashion. However, it is correct only for tree-shaped constraint networks.

In this paper, we show how to extend the algorithm to arbitrary topologies using a pseudotree arrangement of the problem graph, and report our experimental results. The algorithm is formulated for optimization problems, but can be easily applied to satisfaction problems by having relations with utility either 0 (for allowed tuples) or negative values (for disallowed tuples). Utility maximization produces a solution if there is an assignment with utility 0.

The rest of this paper is structured as follows: Section 2 presents the definitions and the notation we use, Section 3 presents an optimization procedure for trees, Section 4 the optimization for graphs, Section 5 proves the complexity to be equal to the induced width, Section 6 compares theoretically our algorithm with other approaches, Section 7 presents experimental results, and we conclude in Section 8.

2 Definitions & notation

A discrete *multiagent constraint optimization problem* (MCOP) is a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ such that:

- $\mathcal{X} = \{X_1, \dots, X_m\}$ is the set of variables/agents;

- $\mathcal{D} = \{d_1, \dots, d_m\}$ is a set of domains of the variables, each given as a finite set of possible values.
- $\mathcal{R} = \{r_1, \dots, r_p\}$ is a set of relations, where a relation r_i is a function $d_{i1} \times \dots \times d_{ik} \rightarrow \mathbb{R}^+$ which denotes how much utility is assigned to each possible combination of values of the involved variables.

In this paper we deal with unary and binary relations, being well-known that higher arity relations can also be expressed in these terms with little modifications. In a MCOP, any value combination is allowed; the goal is to find an assignment $\mathcal{X}^*$ for the variables X_i that maximizes the sum of utilities.

For a node X_k , we define $R_k(X_j) =$ the relation(s) between X_k and its neighbor X_j .

3 Distributed constraint optimization for tree-structured networks

For tree-structured networks, polynomial-time complete optimization methods have been developed (e.g. the sum-product algorithm [Kschischang *et al.*, 2001] and the *DTREE* algorithm from [Petcu and Faltings, 2004]).

In *DTREE*, the agents send *UTIL* messages (utility vectors) to their parents. A child X_l of node X_k would send X_k a vector of the optimal utilities $u_{X_l}^*(v_k^j)$ that can be achieved by the subtree rooted at X_l plus $R_l(X_k)$, and are compatible with each value v_k^j of X_k (such a vector has $|dom(X_k)|$ values).

For the leaf nodes it is immediate to compute these valuations by just inspecting the constraints they have with their single neighbors, so they initiate the process. Then each node X_i relays these messages according to the following process:

- Wait for *UTIL* messages from all children. Since all of the respective subtrees are disjoint, by summing them up, X_i computes how much utility each of its values gives for the whole subtree rooted at itself. This, together with the relation(s) between X_i and its parent X_j , enables X_i to compute exactly how much utility can be achieved by the entire subtree rooted at X_i , taking into account compatibility with each of X_j 's values. Thus, X_i can send to X_j its *UTIL* message. X_i also stores its optimal values corresponding to each value of X_j .
- If root node, X_i can compute the optimal overall utility corresponding to each one of its values (based on all the incoming *UTIL* messages), pick the optimal one, and send a *VALUE* message to its children, informing them about its decision.

Upon receipt of the *VALUE* message from its parent, each node is able to pick the optimal value for itself (as the previously stored optimal value corresponding to the value its parent has chosen), and pass it on to its children. At this point, the algorithm is finished for X_i .

The correctness of this algorithm was shown in the original paper, as well as the fact that it requires a linear number of messages, and linear memory.

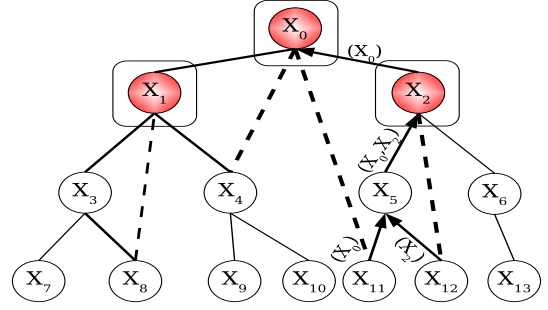


Figure 1: Example of a pseudotree arrangement.

4 Distributed constraint optimization for general networks

To apply a *DTREE*-like algorithm to a cyclic graph, we first need to arrange the graph as a pseudotree (it is known that this arrangement is possible for any graph).

4.1 Pseudotrees

Definition 1 A pseudo-tree arrangement of a graph G is a rooted tree with the same vertices as G and the property that adjacent vertices from the original graph fall in the same branch of the tree (e.g. X_0 and X_{11} in Figure 1).

Pseudotrees have already been investigated as a means to boost search ([Freuder, 1985; Freuder and Quinn, 1985; Dechter, 2003; Schiex, 1999]). The main idea with their use in search, is that due to the relative independence of nodes lying in different branches of the pseudotree, it is possible to perform search in parallel on these independent branches.

Figure 1 shows an example of a pseudotree that we shall refer to in the rest of this paper. It consists of *tree edges*, shown as solid lines, and *back edges*, shown as dashed lines, that are not part of the spanning tree (e.g. $8-1$, $12-2$, $4-0$). We call a path in the graph that is entirely made of tree edges, a *tree-path*. A *tree-path associated with a back-edge* is the tree-path connecting the two nodes involved in the back-edge (please note that since our arrangement is a pseudotree, such a tree path is always included in a branch of the tree). For each back-edge, the higher node involved in that back-edge is called the *back-edge handler* (e.g. the dark nodes 0, 1 and 2). We also define:

- $P(X)$ - the *parent* of a node X : the single node higher in the hierarchy of the pseudotree that is connected to the node X directly through a tree edge (e.g. $P(X_4) = X_1$)
- $C(X)$ - the *children* of a node X : the set of nodes lower in the pseudotree that are connected to the node X directly through tree edges (e.g. $C(X_1) = \{X_3, X_4\}$)
- $PP(X)$ - the *pseudo-parents* of a node X : the set of nodes higher in the pseudotree that are connected to the node X directly through back-edges ($PP(X_8) = \{X_1\}$)
- $PC(X)$ - the *pseudo-children* of a node X : the set of nodes lower in the hierarchy of the pseudotree that are connected to the node X directly through back-edges (e.g. $PC(X_0) = \{X_4, X_{11}\}$)

As it is already known, a DFS (depth-first search) tree is also a pseudotree (although the inverse does not always hold). So, a DFS tree obtained from the DFS traversal of the graph starting from one of the nodes (chosen through a distributed leader election algorithm) will do just fine. Due to the lack of space we do not present here a procedure for the creation of a DFS tree, and refer the reader to techniques like [Barbosa, 1996; Hamadi *et al.*, 1998].

4.2 The DPOP algorithm

Our algorithm has 3 phases. First, the agents establish the pseudotree structure (see section 4.1) to be used in the following two phases. The next two phases are the *UTIL* and *VALUE* propagations, which are similar to the ones from *DTREE* - section 3. Please refer to Algorithm 1 for a formal description of the algorithm, and to the rest of this section for a detailed description of the *UTIL* and *VALUE* phases.

UTIL propagation

As in *DTREE*, the *UTIL* propagation starts from the leaves of the pseudotree and propagates up the pseudotree, only through the tree edges. It is easy for an agent to identify whether it is a leaf in the pseudotree or not: it must have a single tree edge (e.g. X_7 to X_{13} in Figure 1).

In a tree network, a *UTIL* message sent by a node to its parent is dependent only on the subtree rooted at the respective node (no links to other parts of the tree), and the constraint between the node and its parent. For example, consider the message ($X_6 \rightarrow X_2$). This message is clearly dependent only on the target variable X_2 , since there are no links between X_6 or X_{13} and any node above X_2 .

In a network with cycles (each back-edge in the pseudotree produces a cycle), a message sent from a node to its parent may also depend on variables above the parent. This happens when there is a back-edge connecting the sending node with such a variable. For example, consider the message ($X_8 \rightarrow X_3$) in Figure 1. We see that the utilities that the subtree rooted at X_8 can achieve are not dependent only on its parent X_3 (as for $X_6 \rightarrow X_2$). As X_8 is connected with X_1 through the backedge $X_8 \rightarrow X_1$, X_8 must take into account this dependency when sending its message to X_3 .

This is where the dynamic programming approach comes into play: X_8 will compute the optimal utilities its subtree can achieve for each value combination of the tuple $\langle X_3, X_1 \rangle$. It will then assemble a message as a hypercube with 2 dimensions (one for the target variable X_3 and one for the back-edge handler X_1), and send it to X_3 (see Table 1).

This is the key difference between *DTREE* and *DPOP*: messages travelling through the network in *DTREE* always have a single dimension (they are linear in the domain size of the target variable), whereas in *DPOP*, messages have multiple dimensions (one for the target variable, and another one for each context variable).

Combining messages - dimensionality increase

Let us consider this example: X_5 receives 2 messages from its children X_{11} and X_{12} ; the message from X_{11} has X_0 as context, and the one from X_{12} has X_2 as context. Both have one dimension for X_5 (target variable) and one dimension for their context variable (X_0 and X_2 respectively), therefore,

Algorithm 1: *DPOP - Distributed pseudotree-optimization procedure for general networks.*

```

1: DPOP( $\mathcal{X}, \mathcal{D}, \mathcal{R}$ )
   Each agent  $X_i$  executes:
2:
3: Phase 1: pseudotree creation
4: elect leader from all  $X_j \in \mathcal{X}$ 
5: elected leader initiates pseudotree creation
6: afterwards,  $X_i$  knows  $P(X_i)$ ,  $PP(X_i)$ ,  $C(X_i)$  and  $PC(X_i)$ 
7: Phase 2: UTIL message propagation
8: if  $|Children(X_i)| == 0$  (i.e.  $X_i$  is a leaf node) then
9:    $UTIL_{X_i}(P(X_i)) \leftarrow \text{Compute\_utils}(P(X_i), PP(X_i))$ 
10:  Send_message( $P(X_i)$ ,  $UTIL_{X_i}(P(X_i))$ )
11: activate UTIL_Message_handler()
12: Phase 3: VALUE message propagation
13: activate VALUE_Message_handler()
14: END ALGORITHM
15:
16: UTIL_Message_handler( $X_k, UTIL_{X_k}(X_i)$ )
17: store  $UTIL_{X_k}(X_i)$ 
18: if  $Parent(X_i) == \text{null}$  (that means  $X_i$  is the root) then
19:    $v_i^* \leftarrow \text{Choose\_optimal}(\text{null})$ 
20:   Send VALUE( $X_i, v_i^*$ ) to all  $C(X_i)$ 
21: else
22:    $UTIL_{X_i}(P(X_i)) \leftarrow \text{Compute\_utils}(P(X_i), PP(X_i))$ 
23:   Send_message( $P(X_i)$ ,  $UTIL_{X_i}(P(X_i))$ )
24: return
25:
26:
27: VALUE_Message_handler( $VALUE_{P(X_i)}^{X_i}$ )
28: add all  $X_k \leftarrow v_k^* \in VALUE_{P(X_i)}^{X_i}$  to agent.view
29:  $X_i \leftarrow v_i^* = \text{Choose\_optimal}(\text{agent.view})$ 
30: Send VALUE_{X_i}^{X_i} to all  $X_l \in C(X_i)$ 
31:
32: Choose\_optimal(agent.view)
33:
34:    $v_i^* \leftarrow \text{argmax}_{v_i} \sum_{X_l \in C(X_i)} UTIL_{X_l}(v_i, \text{agent.view})$ 
35:
36: return  $v_i^*$ 
37:
38: Compute\_utils( $P(X_i)$ ,  $PP(X_i)$ )
39: for all combinations of values of  $X_k \in PP(X_i)$  do
40:   let  $X_j$  be Parent( $X_i$ )
41:   similarly to DTREE, compute a vector  $UTIL_{X_i}(X_j)$ 
42:   of all  $\{Util_{X_i}(v_i^*(v_j), v_j) | v_j \in Dom(X_j)\}$ 
43: assemble a hypercube  $UTIL_{X_i}(X_j)$  out of all these
44: vectors (totaling  $|PP(X_i)| + 1$  dimensions).
45: return  $UTIL_{X_i}(X_j)$ 

```

$X_8 \rightarrow X_3$	$X_3 = v_3^0$	$X_3 = v_3^1$	...	$X_3 = v_3^{m-1}$
$X_1 = v_1^0$	$u_{X_8}^*(v_1^0)$	$u_{X_8}^*(v_1^1)$	...	$u_{X_8}^*(v_1^{m-1})$
...	...	...	...	...
$X_1 = v_1^{n-1}$	$u_{X_8}^*(v_1^{n-1})$	$u_{X_8}^*(v_1^{n-1})$	...	$u_{X_8}^*(v_1^{n-1})$

Table 1: *UTIL* message sent from X_8 to X_3 , in Figure 1

their dimensionality is 2. X_5 needs to send out its message to its parent (X_2). X_5 considers all possible values of X_2 , and for each one of them, all combinations of values of the context variables (X_0 and X_2) and X_5 are considered; the values of X_5 are always chosen such that the optimal utilities for each tuple $\langle X_0 \times X_2 \times X_5 \rangle$ are achieved. Note that since X_2 is both a context variable and the target variable, the message collapses to 2 dimensions, not 3.

One can think of this process as the cross product of messages $X_{11} \rightarrow X_5$ and $X_{12} \rightarrow X_5$ resulting in a hypercube with dimensions X_0 , X_2 and X_5 , followed by a projection on the X_5 axis, which retains the optimal utilities for the tuples $\langle X_0 \times X_2 \rangle$ (optimizing w.r.t. X_5 given X_0 and X_2).

Collapsing messages - dimensionality decrease

Whenever a multi-dimensional *UTIL* message reaches a target variable that occupies one dimension in the message (a back-edge handler), the target variable optimizes itself out of the context, and the outgoing message loses the respective dimension.

We can take the example of X_1 , which is initially present in the context of the message $X_8 \rightarrow X_3$: once the message arrives at X_1 , since X_1 does not have any more influence on the upper parts of the tree, X_1 can "optimize itself away" by simply choosing the best value for itself, for each value of its parent X_0 (the normal *DTREE* process). Thus, one can see that a back edge handler (X_1 in our case) appears as an extra dimension in the messages travelling from the lower end of the back edge (X_8) to itself, through the tree path associated with the back edge ($X_8 \rightarrow X_3 \rightarrow X_1$).

VALUE propagation

The *VALUE* phase is similar to *DTREE*. Now, in addition to its parent's value, the $VALUE_{P(X_j)}^{X_j}$ message a node X_j receives from its parent also contains the values of all the variables that were present in the context of X_j 's *UTIL* message for its parent. E.g.: X_0 sends X_2 $VALUE_0^2(X_0 \leftarrow v_0^*)$, then X_2 sends X_5 $VALUE_2^5(X_0 \leftarrow v_0^*, X_2 \leftarrow v_2^*)$, and X_5 sends X_{11} $VALUE_5^{11}(X_0 \leftarrow v_0^*, X_5 \leftarrow v_5^*)$.

5 Complexity analysis

By construction, the number of messages our algorithm produces is linear: there are $n - 1$ *UTIL* messages - one through each tree-edge (n is the number of nodes in the problem), and m linear size *VALUE* messages - one through each edge (m is the number of edges). The DFS construction also produces a linear number of messages (good algorithms require $2 \times m$ messages). Thus, the complexity of this algorithm lies in the size of the *UTIL* messages.

Theorem 1 *The largest UTIL message produced by Algorithm 1 is space-exponential in the width of the pseudotree induced by the DFS ordering used.*

PROOF. Dechter ([Dechter, 2003], chapter 4, pages 86-88) describes the *fill-up method* for obtaining the *induced width*. First, we build the *induced graph*: we take the DFS traversal of the pseudotree as an ordering of the graph and process the nodes recursively (bottom up) along this order. When a node is processed, all its parents are connected (if not already

connected). The *induced width* is the maximum number of parents of any node in the induced graph.

It is shown in [Dechter, 2003] that the width of a tree (no back-edges) is 1. Actually the back-edges are the ones that influence the width: a single backedge produces an induced width of 2. From the construction of the induced tree, it is easy to see that several backedges produce increases in the width only when their tree-paths overlap on at least one edge, and their respective handlers are different; otherwise their effects on the width do not combine. Thus, the width is given by the size of the maximal set of back-edges which have overlapping tree-paths and distinct handlers.

During the *UTIL* propagation, the message size varies; the largest message is the one with the most dimensions. We have seen that a dimension X_i is added to a message when a back-edge with X_i as a handler is first encountered in the propagation, and travels through the tree-path associated with the back-edge. It is then eliminated by projection when the message arrives at X_i . The maximal dimensionality is therefore given by the maximal number of overlaps of tree-paths associated with back-edges with distinct handlers.

We have thus shown that the maximal dimensionality is equal to the induced width. $\square$

Exponential size messages are not necessarily a problem in all setups (depending on the resources available and on the induced width - low width problems generate small messages!)

However, when the maximum message size is limited, one can serialize big messages by letting the back-edge handlers ask explicitly for valuations for each one of their values *sequentially*, so each message can have customizable size.

A workaround against exponential memory is possible by renouncing exactness, and propagating valuations for the best/worst value combinations (upper/lower bounds) instead of all combinations.

6 Comparison with other approaches

Schiex [Schiex, 1999] notes the fact that so far, pseudotree arrangements have been mainly used for *search* procedures (essentially backtrack-based search, or branch-and-bound for optimization). As good examples, see the Distributed Depth-first Branch and Bound (DDBB), Distributed Iterative Deepening (DID), ADOPT, Synchronous Branch and Bound (SBB) and Iterative Distributed Breakout (IDB). All these procedures have a worst case complexity exponential in the depth of the pseudotree arrangement (basically because all the variables on the longest branch from root to a leaf have to be instantiated sequentially, and all their value combinations tried out). It was shown in [Bayardo and Miranker, 1995] that there are ways to obtain shallow pseudotrees (within a logarithmic factor of the induced width), but these require intricate heuristics like the ones from [Freuder and Quinn, 1985; Maheswaran *et al.*, 2004], which have not yet been adapted to a distributed setting, as also noted by the authors of the second paper.

In contrast, our approach exhibits a worst case complexity exponential in the *width of the graph* induced by the pseudotree ordering. Arnborg shows in [Arnborg, 1985] that finding a min-width ordering of a graph is NP-hard; however,

the DFS traversal of the graph has the advantage that it produces a good approximation, and is easy to implement in a distributed context. This, coupled with the fact that the depth of the pseudotree is irrelevant to the complexity, means that our algorithm works well with a simple DFS ordering. To see this fundamental difference between the two approaches, consider a problem that is a ring with n nodes. A DFS ordering of such a graph would yield a pseudotree with height n , and one back edge, thus the induced width is 2. A backtracking algorithm is time exponential in n , whereas our algorithm is linear, with message size $O(|d|^2)$. Since the exponential complexity translates directly in the explosion of the number of messages exchanged, these backtracking-based algorithms have not yet been applied to large systems.

Furthermore, it was shown by Dechter in [Dechter and Fattah, 2001] that the induced width is always less than or at most equal with the pseudotree height; thus we can conclude theoretically that our algorithm will always do at least as well as a pseudotree backtrack-based algorithm on the same pseudotree ordering. However, it is only fair to say that our approach can generate very big messages in the worst case, so one has to find a proper tradeoff between the number and the size of the messages transmitted through the system.

7 Experimental evaluation

Usual performance metrics for distributed algorithms are the number of messages and the number of synchronous cycles required to find the optimal solution. Both are linear in our case. For the number of messages, see section 5; the number of synchronous cycles is two times the height of the pseudotree (one *UTIL* propagation, and one *VALUE* propagation). We also introduce the maximal message size as a metric.

7.1 Sensor networks

One of our experimental setups is the sensor grid testbed from [Bejar *et al.*, 2005]. Briefly, there is a set of targets in a sensor field, and the problem is to allocate 3 different sensors to each target. This is a NP-complete resource allocation problem.

In [Bejar *et al.*, 2005], random instances are solved by AWC (a complete algorithm for constraint *satisfaction*). The problems are relatively small (100 sensors and maximum 18 targets, beyond which the problems become intractable). Our initial experiments with this setup solve to optimality problems in a grid of 400 sensors, with up to 40 targets.

Another setup is the one from [Maheswaran *et al.*, 2004], where there are corridors composed of squares which indicate areas to be observed. Sensors are located at each vertex of a square; in order for a square to be "observed", all 4 sensors in its vertices need to be focused on the respective square. Depending on the topology of the grid, some sensors are shared between several squares, and they can observe only one of them at a time. The authors test 4 improved versions of ADOPT (current state of the art for MCOP) on 4 different scenarios, where the corridors have the shapes of capital letters L, Z, T and H. Their results and a comparison with *DPOP* are in Table 2. One can see the dramatic reduction of the number of messages required (in some cases orders of magnitude), even for these very small problem instances (16

Algo/Scenario	Test L	Test Z	Test T	Test H
MCN , No Pass	626.4	1111.64	1841.28	1898.04
MLSP, No Pass	597.88	663.32	477.56	679.36
MCN , Pass	95.67	101.90	94.93	258.07
MLSP , Pass	81.77	91.5	107.77	255.2
DPOP	30	30	18	30

Table 2: DPOP vs 4 ADOPT versions: number of messages in sensor allocation problems.

variables). The explanation is that our algorithm always produces a linear number of messages.

Regarding the size of the messages: these problems have graphs with very low induced widths (2), basically given by the intersections between corridors. Thus, our algorithm employs linear messages in most of the parts of the problems, and only in the intersections are created 2 messages with 2 dimensions (in this case with 64 values each).

This fact gives our algorithm the ability to solve arbitrarily large instances of this particular kind of real-world problems.

7.2 Meeting scheduling

We experimented with distributed meeting scheduling in an organization with a hierarchical structure (a tree with departments as nodes, and a set of agents working in each department). The CSP model is the PEAV model from [Maheswaran *et al.*, 2004]. Each agent has multiple variables: one for the start time of each meeting it participates in, with 8 timeslots as values. Mutual exclusion constraints are imposed on the variables of an agent, and equality constraints are imposed on the corresponding variables of all agents involved in the same meeting. Private, unary constraints placed by an agent on its own variables show how much it values each meeting/start time. Random meetings are generated, each with a certain utility for each agent. The objective is to find the schedule that maximizes the overall utility.

Table 3 shows how our algorithm scales up with the size of the problems. Notice that the total number of messages includes the *VALUE* messages (linear size), and that due to the fact that intra-agent subproblems are denser than the rest of the problem, high-dimensional messages are likely to be virtual, intra-agent messages (not actually transmitted over the network). To our knowledge, these are by far the largest optimization problems solved with a complete, distributed algorithm (200 agents, 101 meetings, 270 variables, 341 constraints). The largest reported previous experiment is [Maheswaran *et al.*, 2004], with 33 agents, 12 meetings, 47 variables, 123 constraints, solved using *ADOPT*.

8 Conclusions and future work

We presented in this paper a new complete method for distributed constraint optimization. This method is a utility-propagation method that extends tree propagation algorithms like the sum-product algorithm or *DTREE* to work on arbitrary topologies using a pseudotree structure. It requires a *linear number of messages*, the largest one being exponential in the induced width along the particular pseudotree cho-

Agents	30	40	70	100	200
Meetings	14	15	34	50	101
Variables	44	50	112	160	270
Constraints	52	60	156	214	341
Messages	95	109	267	373	610
Max message size	512	4096	32k	256k	256k
Cycles	30	32	70	86	96

Table 3: DPOP tests on meeting scheduling.

sen. This method reduces the complexity from dom^n (standard backtracking) to dom^w , where n =number of nodes in the problem, dom bounds the domain size and w =the induced width along the particular pseudotree chosen. For loose problems, $n \gg w$ holds and our method retains the advantage of a linear number of messages (in practice even orders of magnitude fewer messages than the other approaches), while preserving a small message size. In real world scenarios, sending a few larger messages is preferable to sending a lot of small messages because of the much lower overheads implied (differences can go up to orders of magnitude speedups). Our experiments show that our method is the first one to be able to handle effectively arbitrarily large instances of a number of practical problems while using a linear number of messages.

Finding the minimum width pseudotree is an NP-complete problem, so in our future work we will investigate heuristics for finding low width pseudotrees.

9 Acknowledgements

We would like to thank Rina Dechter and Radu Marinescu for insightful discussions, Jonathan Pearce/TEAMCORE for experimental data from sensor networks and meeting scheduling simulations, and Michael Schumacher for valuable comments on an early version of this paper. This work has been funded by the Swiss National Science Foundation under contract No. 200020-103421/1.

References

- [Arnborg, 1985] S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, 25(1):2–23, 1985.
- [Barbosa, 1996] Valmir Barbosa. *An Introduction to Distributed Algorithms*. The MIT Press, 1996.
- [Bayardo and Miranker, 1995] Roberto Bayardo and Daniel Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI-95*, Montreal, Canada, 1995.
- [Bejar et al., 2005] Ramon Bejar, Cesar Fernandez, Magda Valls, Carmel Domshlak, Carla Gomes, Bart Selman, and Bhaskar Krishnamachari. Sensor networks and distributed CSP: Communication, computation and complexity. *Artificial Intelligence*, 161(1-2):117–147, 2005.
- [Dechter and Fattah, 2001] Rina Dechter and Yousri El Fattah. Topological parameters for time-space tradeoff. *Artificial Intelligence*, 125(1-2):93–118, 2001.
- [Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
- [Freuder and Quinn, 1985] Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence, IJCAI-85*, pages 1076–1078, Los Angeles, CA, 1985.
- [Freuder, 1985] Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(14):755–761, 1985.
- [Hamadi et al., 1998] Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *ECAI-98*, pages 219–223, 1998.
- [Kasif, 1986] Simon Kasif. On the parallel complexity of some constraint satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-86*, pages 349–353, Philadelphia, PA, 1986.
- [Kschischang et al., 2001] Frank R. Kschischang, Brendan Frey, and Hans Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions On Information Theory*, 47(2):498–519, FEBRUARY 2001.
- [Maheswaran et al., 2004] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking DCOP to the realworld: Efficient complete solutions for distributed multi-event scheduling. In *AAMAS-04*, 2004.
- [Meisels and Zivan, 2003] Amnon Meisels and Roie Zivan. Asynchronous forward-checking on DisCSPs. In *Proceedings of the Distributed Constraint Reasoning Workshop, IJCAI 2003, Acapulco, Mexico*, 2003.
- [Modi et al., 2003] P. J. Modi, W. M. Shen, and M. Tambe. An asynchronous complete method for distributed constraint optimization. In *Proc. AAMAS*, 2003.
- [Petcu and Faltings, 2004] Adrian Petcu and Boi Faltings. A distributed, complete method for multi-agent constraint optimization. In *CP 2004 - Fifth International Workshop on Distributed Constraint Reasoning (DCR2004) in Toronto, Canada*, September 2004.
- [Schiex, 1999] Thomas Schiex. A note on CSP graph parameters. Technical Report 03, INRA, 07 1999.
- [Silaghi et al., 2000] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *AAAI/IAAI*, pages 917–922, Austin, Texas, 2000.
- [Yokoo et al., 1992] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.
- [Yokoo et al., 1998] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem - formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.

Breaking symmetries in all different problems

Jean-François Puget

ILOG

9 Avenue de Verdun

94253 Gentilly Cedex, France

puget@ilog.fr

Abstract

Adding symmetry breaking constraints is one of the oldest ways of breaking variable symmetries for CSPs. For instance, it is well known that all the symmetries for the pigeon hole problem can be removed by ordering the variables. We have generalized this result to all CSPs where the variables are subject to an all different constraint. In such case it is possible to remove all variable symmetries with a partial ordering of the variables. We show how this partial ordering can be automatically computed using computational group theory (CGT). We further show that partial orders can be safely used together with the GE-tree method of [Roney-Dougal *et al.*, 2004]. Experiments show the efficiency of our method.

1 Introduction

A symmetry for a Constraint Satisfaction Problem (CSP) is a mapping of the CSP onto itself that preserves its structure as well as its solutions. If a CSP has some symmetries, it may be the case that all symmetrical variants of every dead end encountered during the search must be explored before a solution can be found. Even if the problem is easy to solve, all symmetrical variants of a solution are also solutions, and listing all of them may just be impossible in practice. Those observations have triggered a lot of interest for the detection and removal of symmetries in the constraint programming community. Adding symmetry breaking constraints is one of the oldest ways of breaking variable symmetries for (CSPs). For instance, it is shown in [Crawford *et al.*, 1996] that all variable symmetries could be broken by adding one lexicographical ordering constraint per symmetry. Unfortunately, this method is not tractable in general, as there may be an exponential number of symmetries. It has been shown that in general there is no way to break all symmetries of a problem with a polynomial number of constraints [Roy and Luks, 2004]. In [Flener *et al.*, 2002], a linear number of constraints are used to break symmetries for matrix problems. As expected, since there are a polynomial number of constraints, not all symmetries are broken. However, a polynomial number of constraints may be sufficient for breaking all symmetries in some special cases. For instance, in [Puget, 1993],

it is shown that a linear number of constraints can break all symmetries for the pigeon hole problem: one simply needs to order the variables. In this paper we consider a more general class of problems: *all different problems*. These are CSPs such that the variables are subject to an all different constraint among other constraints. We show in section 4 that for such CSPs, all variable symmetries can be broken with at most $n - 1$ binary constraints, where n is the number of variables.

In [Roney-Dougal *et al.*, 2004] a general purpose method for breaking all value symmetries is given: the GE-tree method. We show in section 5 that this method can be safely combined with symmetry breaking constraints, under some conditions on the order in which the search tree is traversed.

In section 6, we apply our method to some complex CSPs. We summarize our findings and discuss some possible generalizations in section 7.

2 Symmetries, Graphs and CSPs

The symmetries we consider are permutations, i.e. one to one mappings (bijections) from a finite set onto itself. Without loss of generality, we can consider permutations of I^n , where I^n is the set of integers ranging from 0 to $n - 1$. For instance, we can label the variables of a graph with integers, such that any variable symmetry is completely described by a permutation of the labels of its variables. This is formalized as follows.

2.1 Computational Group Theory

Let S^n be the set of all permutations of the set I^n . The image of i by the permutation σ is denoted i^σ . A permutation $\sigma \in S^n$ is fully described by the vector $[0^\sigma, 1^\sigma, \dots, (n-1)^\sigma]$. The product of two permutations σ and θ is defined by $i^{(\sigma\theta)} = (i^\sigma)^\theta$.

Given $i \in I^n$ and a permutation group $G \subseteq S^n$, the *orbit* of i in G , denoted i^G , is the set of elements to which i can be mapped to by an element of G :

$$i^G = \{i^\sigma \mid \sigma \in G\}$$

Given $i \in I^n$ and a permutation group $G \subseteq S^n$, the *stabilizer* of i in G , denoted i_G , is the set of permutations of G that leave i unchanged:

$$i_G = \{\sigma \in G \mid i^\sigma = i\}$$

2.2 CSP and symmetries

A *constraint satisfaction problem* $\mathcal{P}$ (CSP) with n variables is a triple $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$ is a finite set of variables $(v_i)_{i \in I^n}$, $\mathcal{D}$ a finite set of finite sets $(\mathcal{D}_i)_{i \in I^n}$, and every constraint in $\mathcal{C}$ is a subset of the cross product $\bigotimes_{i \in I^n} \mathcal{D}_i$. Without loss of generality, we can assume that $\mathcal{D}_i \subseteq I^k$ for some k .

An *assignment* is a member of $\mathcal{S}$, i.e. a vector of values $(a_i)_{i \in I^n}$ such that $a_i \in \mathcal{D}_i$ for all $i \in I^n$, and is denoted $(v_i = a_i)_{i \in I^n}$. A *partial assignment* is sub vector of an assignment.

A *solution* to $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ is an assignment that is consistent with every member of $\mathcal{C}$.

Given a permutation σ of I^n , we define a variable permutation on (partial) assignments as follows:

$$((v_i = a_i)_{i \in I^n})^\sigma = ((v_{i\sigma} = a_i)_{i \in I^n})$$

Such permutation is called a *variable symmetry* if it maps solutions to solutions.

Given a permutation θ of I^k , we define a value permutation on (partial) assignments as follow:

$$((v_i = a_i)_{i \in I^n})^\theta = ((v_i = a_i^{\theta^{-1}})_{i \in I^n})$$

Such permutation is called a *value symmetry* if it maps solutions to solutions.

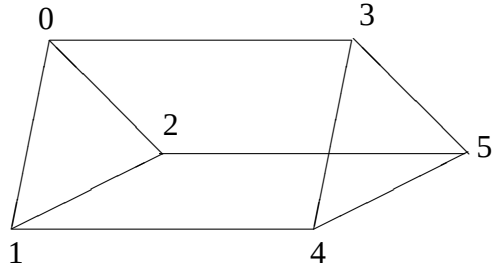
2.3 A graph coloring example

Let us introduce an example that will be used throughout the paper. We say that a graph with m edges is *graceful* if there exists a labeling f of its vertices such that:

- $0 \leq f(i) \leq m$ for each vertex i ,
- the set of values $f(i)$ are all different,
- the set of values $abs(f(i), f(j))$ for every edge (i, j) are all different.

A straightforward translation into a CSP exists where there is a variable v_i for each vertex v_i , see [Lustig and Puget, 2001]. The variable symmetries of the problem are induced by the automorphism of the graph. There is one value symmetry, which maps v to $m - v$. More information on symmetries in graceful graphs is available in [Petrie and Smith, 2003], [Petrie, 2004].

Let us consider the following graph $K_3 \times P_2$:



The group of variable symmetries of the corresponding CSP is equivalent to the group of symmetries of the graph. Such group can be computed by packages such as Nauty[Mc Kay, 1981]. This group G is:

$$\{[0, 1, 2, 3, 4, 5], [0, 2, 1, 3, 5, 4], [1, 0, 2, 4, 3, 5],$$

$$[1, 2, 0, 4, 5, 3], [2, 0, 1, 5, 3, 4], [2, 1, 0, 5, 4, 3], \\ [3, 4, 5, 0, 1, 2], [3, 5, 4, 0, 2, 1], [4, 3, 5, 1, 0, 2], \\ [4, 5, 3, 1, 2, 0], [5, 3, 4, 2, 0, 1], [5, 4, 3, 2, 1, 0]\}$$

3 Breaking variable symmetries

Without loss of generality, we can assume that domains are subsets of I^k for some k , with the usual ordering on integers.

3.1 Lex leader constraints

Adding constraints is one of the oldest methods for reducing the number of variable symmetries of a CSP [Puget, 1993]. In [Crawford et al., 1996], it is shown that all the variable symmetries of any CSP can be broken by the following constraints.

$$\forall \sigma \in G, \mathcal{V} \preceq \mathcal{V}^\sigma \quad (1)$$

For a given σ , the constraint $(\mathcal{V} \preceq \mathcal{V}^\sigma)$ is semantically equivalent to the disjunction of the constraints:

$$\begin{aligned} v_0 &< v_{0\sigma} \\ v_0 &= v_{0\sigma} \wedge v_1 < v_{1\sigma} \\ &\vdots \\ v_0 &= v_{0\sigma} \wedge \dots \wedge v_{i-1} = v_{(i-1)\sigma} \wedge v_i < v_{i\sigma} \\ &\vdots \\ v_0 &= v_{0\sigma} \wedge \dots \wedge v_{n-2} = v_{(n-2)\sigma} \wedge v_{n-1} < v_{(n-1)\sigma} \\ v_0 &= v_{0\sigma} \wedge \dots \wedge v_{n-2} = v_{(n-2)\sigma} \wedge v_{n-1} = v_{(n-1)\sigma} \end{aligned}$$

If the last constraint is omitted, the set of constraints is denoted $\mathcal{V} \prec \mathcal{V}^\sigma$.

In our example, the constraints given by [Crawford et al., 1996] are

$$\begin{aligned} (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_0, v_1, v_2, v_3, v_4, v_5) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_0, v_2, v_1, v_3, v_5, v_4) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_1, v_0, v_2, v_4, v_3, v_5) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_1, v_2, v_0, v_4, v_5, v_3) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_2, v_0, v_1, v_5, v_3, v_4) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_2, v_1, v_0, v_5, v_4, v_3) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_3, v_4, v_5, v_0, v_1, v_2) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_3, v_5, v_4, v_0, v_2, v_1) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_4, v_3, v_5, v_1, v_0, v_2) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_4, v_5, v_3, v_1, v_2, v_0) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_5, v_3, v_4, v_2, v_0, v_1) \\ (v_0, v_1, v_2, v_3, v_4, v_5) &\preceq (v_5, v_4, v_3, v_2, v_1, v_0) \end{aligned}$$

3.2 A polynomial number of constraints

The number of constraints (1) can grow exponentially with the number of variables $\mathcal{V}$. Using the fact that the variable are subject to an all different constraint, we can significantly reduce the number of symmetry breaking constraints. Let us consider one of the symmetries of our example, namely:

$$\sigma = [0, 2, 1, 3, 5, 4]$$

The constraint breaking this symmetry is

$$(v_0, v_1, v_2, v_3, v_4, v_5) \preceq (v_0, v_2, v_1, v_3, v_5, v_4)$$

Since $v_0 = v_0$ is trivially true, and since $v_1 = v_2$ cannot be true because of the all different constraint, this constraint can be simplified into:

$$v_1 < v_2$$

This simplification is true in general and can be formalized as follows. Given a permutation σ , let $s(\sigma)$ be the smallest i such that $i^\sigma \neq i$, and let $t(\sigma)$ be equal to $(s(\sigma))^\sigma$.

Lemma 1. *Given a CSP where the variables $\mathcal{V}$ are subject to an all different constraint, and a variable symmetry group G for this CSP, then all variable symmetries can be broken by adding the following constraints:*

$$\forall \sigma \in G, v_{s(\sigma)} < v_{t(\sigma)} \quad (2)$$

Proof. By definition $k^\sigma = k$ for all $k < s(\sigma)$, and $s(\sigma)^\sigma \neq s(\sigma)$. Let us look at the constraint $\mathcal{V} \preceq \mathcal{V}_\sigma$. There is an all different constraint on the variables $\mathcal{V}$, which means that $v_i = v_{i^\sigma}$ if and only if $i^\sigma = i$. In particular, $v_k = v_{k^\sigma}$ for all $k < s(\sigma)$, and $v_{s(\sigma)} \neq v_{(s(\sigma))^\sigma}$. Therefore, only one disjunct for the constraint can be true, namely:

$$v_0 = v_{0^\sigma} \wedge \dots \wedge v_{s(\sigma)-1} = v_{(s(\sigma)-1)^\sigma} \wedge v_{s(\sigma)} < v_{(s(\sigma))^\sigma}$$

Since $k^\sigma = k$ for $k < s(\sigma)$ and $s(\sigma)^\sigma = t(\sigma)$, this can be simplified into $v_{s(\sigma)} < v_{t(\sigma)}$. $\square$

Note that if two permutations σ and θ are such that $s(\sigma) = s(\theta)$ and $t(\sigma) = t(\theta)$, then the corresponding symmetry breaking constraints are identical. Therefore, it is sufficient to state only one symmetry breaking constraints for each pair i, j such that there exists a permutation σ with $i = s(\sigma)$ and $j = t(\sigma)$.

The set of these pairs can be computed using what is known as the Schreier Sims algorithm [Seress, 2003]. This algorithm constructs a stabilizers chain $G_0, G_1, \dots, G_n$ as follows:

$$\begin{aligned} G_0 &= G \\ \forall i \in I^n, G_i &= (i-1)_{G_{i-1}} \end{aligned}$$

By definition,

$$\begin{aligned} G_i &= \{\sigma \in G : 0^\sigma = 1 \wedge \dots \wedge (i-1)^\sigma = i-1\} \\ G_n &\subseteq G_{n-1} \subseteq \dots \subseteq G_1 \subseteq G_0 \end{aligned}$$

The Schreier Sims algorithm also computes set of coset representatives U_i . Those are orbits of i in G_i :

$$U_i = i^{G_i}$$

By definition, U_i is the set of values which i is mapped to by all symmetries in G that leave at least $0, \dots, (i-1)$ unchanged.

From now on, we will assume that all the groups we use are described by a stabilizers chain and coset representatives.

In our example, the stabilizer chain is :

$$\begin{aligned} G_0 &= G \\ G_1 &= 0_{G_0} = \{[0, 1, 2, 3, 4, 5], [0, 2, 1, 3, 5, 4]\} \\ G_2 &= 1_{G_1} = \{[0, 1, 2, 3, 4, 5]\} \end{aligned}$$

All remaining stabilizers G_3, G_4, G_5 are equal to G_2 .

Coset representatives are:

$$\begin{aligned} U_0 &= 0^{G_0} = \{0, 1, 2, 3, 4, 5\} \\ U_1 &= 1^{G_1} = \{1, 2\} \\ U_2 &= 2^{G_2} = \{2\} \\ U_3 &= 3^{G_3} = \{3\} \\ U_4 &= 4^{G_4} = \{4\} \end{aligned}$$

Theorem 2. *Given a CSP with n variables $\mathcal{V}$ such that there exists an all different constraint on these variables, and given coset representatives sets U_i for the variable symmetry group of the CSP, then all the variable symmetries can be broken by at most $n(n-1)/2$ binary constraints. These constraints are given by :*

$$\forall i \in I^n, \forall j \in U_i, i \neq j \rightarrow v_i < v_j \quad (3)$$

Proof. By definition, for each element $j \in U_i$, there exists at least one permutation $\sigma \in G_i$ such that $i^\sigma = j$ and $j = t(\sigma)$. The converse is also true. If there exists a permutation σ such that $i = s(\sigma)$ and that $j = t(\sigma)$, then $j \in U_i$. Therefore, the constraints (2) can be rewritten into:

$$\forall i \in I^n, \forall j \in U_i, i \neq j \Rightarrow v_i < v_j$$

There are $\sum_{i=0}^{n-1} (|U_i| - 1)$ such constraints. All the permutations of G_i leave the numbers $0, \dots, i-1$ unchanged. Therefore U_i is a subset of $\{i, \dots, n-1\}$. Then $|U_i| - 1 \leq n - i - 1$. Therefore, the number of constraints is bounded from above by $\sum_{i=0}^{n-1} (n - i - 1) = n(n-1)/2$. $\square$

In our example, these constraints are :

$$v_0 < v_1, v_0 < v_2, v_0 < v_3, v_0 < v_4, v_0 < v_5, v_1 < v_2$$

Note that some of these constraints are redundant. For instance, the constraint $v_0 < v_2$ is entailed by the first and the last constraints. This remark can be used to reduce the number of constraints as explained in the following section.

3.3 A linear number of constraints

The previous result can be improved by taking into account the transitivity of the $<$ constraints. Given $j \in I^n$, it may be the case that j belongs to several of the sets U_i . In such case, let us define $r(j)$ as the largest i different from j such that j belongs to U_i . If j belongs to no U_i other than U_j , then let $r(j) = j$.

Before stating our main result, let us prove the following.

Lemma 3. *With the above notations, if $j \in U_i$ and $i \neq j$ then $r(j) \in U_i$ and $r(j) < j$*

Proof. Let us assume that $j \in U_i$ and $i \neq j$. By definition of U_i there exists a permutation $\sigma \in G_i$ such that $i^\sigma = j$. Let $k = r(j)$. By definition of $r(j)$, $i \leq k$ and $j \in U_k$. Therefore, there exists a permutation $\theta \in G_k$ such that $k^\theta = j$. Let $\nu = \sigma\theta^{-1}$. Then, $i^\nu = i^{\sigma\theta^{-1}} = j^{\theta^{-1}} = k$. Moreover, $\nu \in G_i$ because $\sigma \in G_i$ and $\theta \in G_k \subseteq G_i$. Therefore,

$k \in U_i$. The fact that $r(j) < j$ is an immediate consequence of the definition of $r(j)$. $\square$

We can now state our main result.

Theorem 4. *With the above notations, given a CSP with n variables $\mathcal{V}$, such that there exists an all different constraint on these variables, then all variable symmetries can be broken by at most $n - 1$ binary constraints. These constraints are given by :*

$$\forall j \in I^n, r(j) \neq j \rightarrow v_{r(j)} < v_j \quad (4)$$

Proof. The number of constraints (4) is at most n by definition. Note that $r(0) = 0$ by definition of r , therefore, the number of constraints is at most $n - 1$. Let us consider one of the constraints of (3). We are given i and j such that $j \in U_i$ and $i \neq j$. We want to prove that the constraint $c = (v_i < v_j)$ is implied by the constraints (4). Let us consider the sequence $(j, r(j), r(r(j)), r(r(r(j))), \dots)$. Let us assume that the sequence never meets i . We have that $j \in U_i$ and $i \neq j$. By application of lemma 3, we get $r(j) \in U_i$ and $r(j) < j$. Since $r(j) \neq i$ by hypothesis, lemma 3 can be applied again. By repeated applications of lemma 3 we construct an infinite decreasing sequence of integers all included in U_i . This is not possible as U_i is finite. Therefore, there exists k such that $i = r^k(j)$. Moreover, we have established $r^k(j) \neq r^{k-1}(j), \dots, r(r(j)) \neq r(j), r(j) \neq j$. Therefore, the constraints $v_{r^k(j)} < v_{r^{k-1}(j)}, \dots, v_{r(r(j))} < v_{r(j)}, v_{r(j)} < v_j$ are constraints of (4). Together they imply $v_{r^k(j)} < v_j$ which is the constraint c . We have proved that the constraints (3) are implied by the constraints (4). Since the set of constraints (4) is a subset of the constraints (3), both sets of constraints are equivalent. Then, by theorem 2, the constraints (4) break all variable symmetries. $\square$

In our example, we get from coset representatives:

$$r(0) = 0, r(1) = 0, r(2) = 1, r(3) = 0, r(4) = 0, r(5) = 0$$

Therefore, the constraints (4) given by theorem 4 are:

$$v_0 < v_1, v_0 < v_3, v_0 < v_4, v_0 < v_5, v_1 < v_2$$

Note that the constraint $v_0 < v_2$ is no longer appearing.

4 Breaking both variable symmetries and value symmetries

In [Roney-Dougal *et al.*, 2004], a general method for breaking all value symmetries is described. This method uses the group of value symmetries of the CSP. We will show that this method can be combined with symmetry breaking constraints when there are both variable symmetries and value symmetries.

4.1 GE-tree and symmetry breaking constraints

We are given a CSP $\mathcal{P}$ with n variables v_i subject to an all different constraint among other constraints. Without loss of generality, we can assume that the domains of the variables are subsets of I^k for some k . It is shown in [Flener *et al.*,

2002] how to transform $\mathcal{P}$ into a new CSP $\mathcal{P}'$ such that all value symmetries of $\mathcal{P}$ become variable symmetries of $\mathcal{P}'$. The idea is to add $n \times k$ additional binary variables x_{ij} (variables with domains equal to $\{0, 1\}$). We also add the following channeling constraints:

$$\forall i \in I^n, j \in I^k, (x_{ij} = 1) \equiv (v_i = j)$$

These constraints state that the variable x_{ij} equals 1 if and only if the variable v_j equals j . Adding these new variables do not change the solutions of the CSP. Moreover, variable symmetries of $\mathcal{P}$ are equivalent to permutations of the rows of the x_{ij} matrix, whereas value symmetries of $\mathcal{P}$ are equivalent to permutations of the columns of the same matrix.

Let us construct the vector X by concatenating the rows of the matrix x_{ij} . Therefore, the variables x_{ij} are ranked in increasing values of i then increasing values of j in the vector X .

Let us consider a value symmetry θ for $\mathcal{P}$. Then θ is a permutation of the matrix columns. This symmetry is broken by the constraint:

$$X \preceq X^\theta \quad (5)$$

Let X_i be the variables in the i -th row of the matrix. The value symmetry θ maps variables in a given row to variables in the same row. This is formalized as follows.

$$\begin{aligned} X_i &= (x_{i0}, x_{i1}, \dots, x_{i(k-1)}) \\ (X^\theta)_i &= (x_{i\theta^{-1}}, x_{i\theta^{-1}-1}, \dots, x_{i(k-1)\theta^{-1}}) \end{aligned}$$

From the definition of $\preceq$, we have that (5) is equivalent to the disjunction of the following constraints:

$$\begin{aligned} X_0 &\prec (X^\theta)_0 \\ X_0 &= (X^\theta)_0 \wedge X_1 \prec (X^\theta)_1 \\ &\vdots \\ X_0 &= (X^\theta)_0 \wedge \dots \wedge X_{i-1} = (X^\theta)_{i-1} \wedge X_i \prec (X^\theta)_i \\ &\vdots \\ X_0 &= (X^\theta)_0 \wedge \dots \wedge X_{n-2} = (X^\theta)_{n-2} \wedge X_{n-1} \prec (X^\theta)_{n-1} \\ X_0 &= (X^\theta)_0 \wedge \dots \wedge X_{n-2} = (X^\theta)_{n-2} \wedge X_{n-1} = (X^\theta)_{n-1} \end{aligned}$$

Let us compare lexicographically X_i with $(X^\theta)_i$. Let a_i be the value assigned to v_i . Then $x_{i(a_i)} = 1$ and $x_{ij} = 0$ for $j \neq a_i$. Similarly, $x_{i(j)\theta^{-1}} = 1$ if and only if $j^{\theta^{-1}} = a_i$, i.e. $a_i^\theta = j$. Therefore, $X_i = (X^\theta)_i$ if and only if $a_i = (a_i)^\theta$, and $X_i \prec (X^\theta)_i$ if and only if $a_i < (a_i)^\theta$.

We then have that (5) is equivalent to the disjunction of the constraints:

$$\begin{aligned} a_0 &< (a_0)^\theta \\ a_0 &= (a_0)^\theta \wedge a_1 < (a_1)^\theta \\ &\vdots \\ a_0 &= (a_0)^\theta \wedge \dots \wedge a_{i-1} = (a_{i-1})^\theta \wedge a_i < (a_i)^\theta \\ &\vdots \\ a_0 &= (a_0)^\theta \wedge \dots \wedge a_{n-2} = (a_{n-2})^\theta \wedge a_{n-1} < (a_{n-1})^\theta \\ a_0 &= (a_0)^\theta \wedge \dots \wedge a_{n-2} = (a_{n-2})^\theta \wedge a_{n-1} = (a_{n-1})^\theta \end{aligned}$$

Let us now consider one of the disjunct, namely:

$$a_0 = (a_0)^\theta \wedge \dots \wedge a_{i-1} = (a_{i-1})^\theta \wedge a_i < (a_i)^\theta$$

This means that θ leaves invariant $a_0, a_1, \dots, a_{i-1}$. In such case a_i must be minimal among the values that any such θ can map it to. We have therefore proved the following result.

Lemma 5. *With the above notations, a_i is the minimum of its orbit in the group of symmetries that leave $a_0, a_1, \dots, a_{i-1}$ unchanged.*

This is equivalent to the GE-tree method for breaking all value symmetries [Roney-Dougal *et al.*, 2004], when the variables and the values are tried in an increasing order during search.

From [Crawford *et al.*, 1996], it is safe to add all possible symmetry breaking constraints (1) on $\mathcal{P}'$. In particular, it is safe to state all the constraints (1) for the variable symmetries of $\mathcal{P}$ together with all the constraints (5). By lemma 5, the set of constraints (5) is equivalent to the GE-tree method for breaking value symmetries. We have just proved the following result.

Theorem 6. *Given a CSP, its group of variable symmetries G_1 , and its group of value symmetries G_2 , then the combination of the GE-tree method for breaking value symmetries with the symmetry breaking constraints (1) computes a set of solutions $\mathcal{S}$ such that:*

$$\forall S \in \text{sol}(\mathcal{P}), \exists \sigma \in G_1, \exists \theta \in G_2, \exists S' \in \mathcal{S}, S^{\sigma\theta} = S'$$

Theorem 4 in section 3 says that the set of all those constraints (1) is equivalent to the constraints (4) when there is an all different constraints on all the variables $\mathcal{V}$. This yields the following result.

Corollary 7. *Given a CSP where the variable are subject to an all different constraint, its group of variable symmetries G_1 , and its group of value symmetries G_2 , then the combination of the GE-tree method for breaking value symmetries with the symmetry breaking constraints (4) computes a set of solutions $\mathcal{S}$ such that:*

$$\forall S \in \text{sol}(\mathcal{P}), \exists \sigma \in G_1, \exists \theta \in G_2, \exists S' \in \mathcal{S}, S^{\sigma\theta} = S'$$

5 Experimental results

We have implemented an algorithm similar to Nauty[Mc Kay, 1981] for computing graph automorphisms, as well as a Schreier Sims algorithm[Seress, 2003]. These have been used in the following examples. In our implementation, we did not fully implement the GE-tree method, because it requires more computational group algorithms than what we have implemented so far. We simply compute the orbits for the group G of value symmetries. Then, only the minimum element of each orbit is left in the domain of the variable v_0 . We will refer to this method as SBC (for Symmetry Breaking Constraints) in order to differentiate it from other methods.

5.1 Graceful graphs

We have tested our approach on the graceful graphs of [Petrie and Smith, 2003]. Variable symmetries are broken by the constraints (4). There is one non trivial value symmetry, which maps a to $e - a$; where e is the number of edges of the graph. Therefore, the orbits for this symmetry are the sets $\{a, e - a\}$, for $0 \leq a \leq e/2$. Therefore, one can restrict the domain of v_0 by keeping one the smallest value in each of these orbits.

For each graph we report the number of solutions of the CSP (sol), the size of the search tree (node) and the time (time) needed to compute all these solutions the running time without symmetry breaking technique (no sym). We also report these figures when the SBC method is used. In this case the running time includes the time needed to perform all the group computations. Running times are measured on a 1.4 GHz Dell Latitude D800 laptop running Windows XP. The implementation is done with ILOG Solver 6.0[ILOG, 2003].

graph	no sym			SBC		
	sol	node	time	sol	node	time
$K_3 \times P_2$	96	1518	0.12	8	83	0.01
$K_4 \times P_2$	1440	216781	13.6	30	1863	0.27
$K_5 \times P_2$	480	34931511	4454	2	53266	6.5
$K_6 \times P_2$				0	1326585	305

Table 1. Computing all solutions for graceful graphs.

The running times are up to 30 times smaller than the ones reported in [Petrie, 2004] for the GAP-SBDD and the GAP-SBDS methods, using a computer about half the speed of ours. This shows that in this example our symmetry breaking constraints are much more efficient than modified search methods. However, we find twice as many more solutions. Let see why on the graph $K_5 \times P_2$. This graph has 10 vertices and 25 edges. We list the values for the variables $v_0, v_1, \dots, v_9$ for the two solutions:

$$(0, 4, 18, 19, 25, 23, 14, 6, 3, 1)$$

$$(0, 6, 7, 21, 25, 24, 22, 19, 11, 2)$$

Let us apply the non trivial value symmetry to the second one. We get:

$$(25, 19, 18, 4, 0, 1, 3, 6, 14, 23)$$

Let us apply the following variable symmetry to it:

$$[4, 3, 2, 1, 0, 9, 8, 7, 6, 5]$$

This yields the first solution!

This example shows that we did not break all symmetries that are a product of a variable symmetry by a value symmetry. This is so despite the fact that all variable symmetries and all value symmetries are broken.

5.2 Most Perfect Magic Squares

Most perfect magic squares, studied in [Ollerenshaw, 1986], are given as an example of a CSP with convoluted variables symmetries in [Roney-Dougal *et al.*, 2004]. The authors decided to use an inverse representation in order to transform variable symmetries into value symmetries. These were

in turn broken with the GE-tree method. In [Ollerenshaw, 1986], it is proven that most perfect magic squares are in a one to one relationship with *reversible squares*. A reversible square of size $n \times n$ (where $n \equiv 0 \pmod{4}$) has entries $1 \dots n^2$ such that (i) the sum or the two entries at diagonally opposite corners of any rectangle or sub-square equals the sum of the other pair of diagonally opposite corners (ii) in each row or column, the sum of the first and last entries equals the sum of the next and the next to last number, etc (iii) diametrically opposed numbers sum to $n^2 + 1$.

Any solution is one of $2^{n+1}((n/2)!)^2$ symmetric equivalent [Ollerenshaw, 1986]. For $n = 16$, this is about $2.13e+14$.

The natural model for this problem has one variable per cell in the square with entries as values. In addition to the above constraints on entries, there is an all different constraint. Therefore, our variable symmetry breaking constraints can be used. We report for various sizes the time used to compute the symmetry breaking constraints as well as the time for finding all non symmetrical solutions with our SBC method. We also report the results of [Roney-Dougal *et al.*, 2004], obtained with GAP-SBDD and with GE-tree on a computer about half the speed of ours. A direct comparison is difficult because they directly search for most perfect magic squares whereas we search for reversible squares. It is worth comparing the time spent in the symmetry computations though, because these deal with the same symmetry group. Our method spends much less time in symmetry computations because this needs to be done only once, before the search starts.

Method	n	sols	sym	search
SBC	4	3	0.01	0.02
	8	10	0.09	0.39
	12	42	0.44	22.2
	16	35	4.6	275.6
GAP-SBDD	4	3	0.3	0.3
	8	10	5.4	125.4
	12	42	2745	12518
GE-tree	4	3	0.2	0.1
	8	10	0.7	90.0
	12	42	29.1	10901.8

6 Discussion

We have established two major results (i) all variable symmetries can be broken by a linear number of binary constraints if there is an all different constraints on all the variables of the CSP (ii) symmetry breaking constraints of [Crawford *et al.*, 1996] can be safely used in conjunction with the GE-tree method of [Roney-Dougal *et al.*, 2004].

Furthermore, these methods can be fully automated using automorphism packages such as Nauty [Mc Kay, 1981] and computational group theory [Seress, 2003]. We have implemented such algorithms. Experiments on complex problems show that these algorithms are quite efficient.

The results described in this paper can be generalized. First of all, theorem 4 is valid for all CSPs where the variables are subject to an all different constraint. It would be interesting to see if similar results can be obtained for other forms of CSPs.

It is worth mentioning that we presented a method for breaking all variable symmetries, and all value symmetries. However, our method does not break products of both kinds of symmetries. It remains to be seen if a simple combination of variable and value symmetry breaking techniques can break all such symmetries.

Acknowledgements

The author would like to thank Marie Puget and the anonymous referees for their remarks. It greatly helped improving the readability of this paper.

References

- [Crawford *et al.*, 1996] Crawford, J., Ginsberg, M., Luks E.M., Roy, A. "Symmetry Breaking Predicates for Search Problems." In proceedings of KR'96, 148-159.
- [Flener *et al.*, 2002] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh.: "Breaking Row and Column Symmetries in Matrix Models." "Proceedings of CP'02, pages 462-476, 2002
- [ILOG, 2003] ILOG: ILOG Solver 6.0. User Manual. ILOG, S.A., Gentilly, France, Septembre 2003
- [Lustig and Puget, 2001] Lustig, I.J., and Puget, J.F. (2001). "Program Does Not Equal Program: Constraint Programming and its Relationship to Mathematical Programming," Interfaces 31(6), 29-53.
- [Mc Kay, 1981] Mc Kay, B.: "Practical Graph Isomorphism" Congr. Numer. 30, 45-87, 1981
- [Ollerenshaw, 1986] Dame Ollerenshaw, K. "On most perfect or complete 8x8 pandiagonal magic squares" Proceedings Royal Society London, 407, 259-281, 1986.
- [Petrie and Smith, 2003] Petrie, K., Smith, B.M. : "Symmetry breaking in graceful graphs." In proceedings of CP'03, LNCS 2833, 930-934, Springer Verlag, 2003.
- [Petrie, 2004] Petrie, K. : "Combining SBDS and SBDD" Technical report APES-86-2004. Available from <http://www.dcs.st-and.ac.uk/~apes/apesreports.html>
- [Puget, 1993] Puget, J.-F.: "On the Satisfiability of Symmetrical Constraint Satisfaction Problems." Proceedings of ISMIS'93 (1993), 350-361.
- [Roney-Dougal *et al.*, 2004] Roney-Dougal C.M., Gent, I.P., Kelsey T., Linton S.: "Tractable symmetry breaking using restricted search trees" To appear in proceedings of ECAI'04.
- [Roy and Luks, 2004] Roy, A., Luks, E.: "The complexity of symmetry-breaking formulas", *Annals of Mathematics and Artificial Intelligence*, 41 (2004), 19-45 (with A. Roy).
- [Seress, 2003] Seress, A.: *Permutation Group Algorithms* Cambridge University Press, 2003.

Efficient belief-state AND–OR search, with application to Kriegspiel

Stuart Russell and Jason Wolfe

Computer Science Division

University of California, Berkeley, CA 94720

russell@cs.berkeley.edu, jawolfe@berkeley.edu

Content areas: Game-playing, uncertainty

Abstract

The paper reports on new algorithms for solving partially observable games. Whereas existing algorithms apply AND-OR search to a tree of black-box belief states, our “incremental” versions treat uncertainty as a new search dimension, examining the physical states within a belief state to construct solution trees incrementally. On a newly created database of checkmate problems for Kriegspiel (a partially observable form of chess), incrementalization yields speedups of two or more orders of magnitude on hard instances.

1 Introduction

“Classical” games, such as chess and backgammon, are fully observable. Partially observable games, despite their greater similarity to the real world, have received less attention in AI. The overall computational task in such games can be divided conceptually into two parts. First, *state estimation* is the process of generating and updating the *belief state*—a representation of the possible states of the world, given the observations to date. Second, *move selection* is the process of choosing a move given the belief state.

Partially observable games are intrinsically more complicated than their fully observable counterparts. At any given point, the set of logically possible states may be very large, rendering both state estimation and move selection intractable. Furthermore, move selection must consider one’s own information state (gathering information is helpful) and the opponent’s information state (revealing information is harmful). Finally, game-theoretically optimal strategies are *randomized* because restricting oneself to deterministic strategies provides additional information to the opponent. The paper by Koller and Pfeffer [1997] provides an excellent introduction to these issues.

In this paper, we focus on a particular subproblem: deciding whether a *guaranteed win* exists—i.e., a strategy that guarantees the optimal payoff within some finite depth, regardless of the true state, for any play by the opponent. This is simpler than the general problem, for two reasons. First, the state estimate need represent only the *logically possible* states, without considering their probabilities. Second, we have the following:

Theorem 1 *A guaranteed win exists from some initial belief state iff it exists against an opponent with full observation.*

This follows from the fact that an opponent, by simply *choosing moves at random*, has a nonzero probability of duplicating any finite behavior of an optimal opponent with full observation. Thus, the *logical possibility* that the solver can guarantee a win within some bounded depth depends only upon the *solver’s* own information about the true state. This implies that neither the opponent’s information state nor randomized strategies need be considered. We will see shortly that the theorem does *not* apply to certain other strategies, which win *with probability 1* but are not *guaranteed*.

Despite these restrictions, the problem is quite general. Indeed, it is isomorphic to that of finding guaranteed plans in nondeterministic, partially observable environments. Our particular focus in this paper is on *Kriegspiel*, a variant of chess in which the opponent’s pieces are *completely invisible*. As the game progresses, partial information is supplied by a referee who has access to both players’ positions. The players, White¹ and Black, both hear every referee announcement. There is no universally accepted set of rules; we adopt the following:

- White may propose to the referee any move that would be a legal chess move on a board containing just White’s pieces; White may also propose any pawn capture.
- If the proposed move is illegal on the board containing both White and Black pieces, the referee announces “Illegal.”² White may then try another move.
- If the move is legal, it is made. Announcements are:
 - If a piece is captured on square *X*: “Capture on *X*.”
 - If Black is now in check: “Check by *D*,” where *D* is one or two of the following directions (from the perspective of Black’s king): Knight, Rank, File, Long Diagonal, and Short Diagonal.
 - If Black has no legal moves: “Checkmate” if Black is in check and “Stalemate” otherwise.
- Finally: “Black to move.”

(Examples of play are given in Section 4. In addition, more details are available online at our Kriegspiel website, www.cs.berkeley.edu/~jawolfe/kriegspiel/.)

One important aspect of Kriegspiel and many other partially observable games and planning domains is the “try-until-feasible” property—a player may attempt any number

¹W.l.o.g., we assume it is White’s turn to move.

²If the move is not legal for White’s pieces alone, or if it has already been rejected on this turn, the referee says “Nonsense.”

of actions until a legal one is found. Because the order of attempts matters, a move choice is actually a plan to try a given *sequence* of potentially legal moves. Hence, the branching factor *per actual move* is the number of such sequences, which can be superexponential in the number of potentially legal moves. We will see that try-until-feasible domains admit certain simplifications that mitigate this branching factor.

Kriegspiel is very challenging for humans. Even for experts, “announced” checkmates are extremely rare except for so-called *material wins*—such as KQR-vs.-K—when one side is reduced to just a king and the other has sufficient (safe) material to force a win. Checkmate is usually “accidental” in the sense that one player mates the other without knowing it in advance.

Several research groups have studied the Kriegspiel checkmate problem. Ferguson [1992] exhibited a randomized strategy for the KBN-vs.-K endgame that wins *with probability 1*; the lone Black king can escape checkmate only by guessing White’s moves correctly infinitely often. Subsequently [1995], he derived a strategy for the KBB-vs.-K endgame that wins with probability $1 - \epsilon$ for any $\epsilon > 0$. In our terminology, these mates are not *guaranteed*, and they do not work if the opponent can see the board. Additional material-win strategies, all deterministic, have been developed [Ciancarini *et al.*, 1997; Bolognesi and Ciancarini, 2003; 2004].

As we explain in Section 4, algorithms for *finding* a checkmate in Kriegspiel involve searching an AND–OR tree whose nodes correspond to belief states. This idea is common enough in the AI literature on partially observable planning search (see, for example, Chapters 3 and 12 of [Russell and Norvig, 2003]). It was proposed for Kriegspiel, and applied to the analogous partially observable variant of Shogi (Japanese chess), by Sakuta and Iida; results for several search algorithms are summarized in Sakuta’s PhD thesis [2001]. Bolognesi and Ciancarini [2004] add heuristic measures of “progress” to guide the tree search, but consider only positions in which the opponent has a lone king. In all of these papers, the initial belief state is determined externally.

The search algorithm developed by Ginsberg [1999] for (partially observable) Bridge play works by sampling the initial complete deal and then solving each deal as a fully observable game. This approach gives a substantial speedup over solving the true game tree, but never acts to gather or hide information (which is essential for domains such as Kriegspiel). The only previous Kriegspiel-playing agent we know of, developed by Parker *et al.* [2005], uses this approach. It keeps track of a sample of its true belief state, and at each point selects the move that would be best if the remainder of the game were played as fully observable chess.

Solving Kriegspiel is an instance of nondeterministic partially observable planning, and could therefore be carried out by symbolic techniques such as the ordered binary decision diagram (OBDD) methods developed by Bertoli *et al.* [2001] or by quantified Boolean formulae (QBF) solvers. Unfortunately, the computational penalty for generating chess moves by symbolic inference methods appears to be around four orders of magnitude [Selman, personal communication].

This paper’s contributions are as follows. Section 2 addresses the problem of state estimation; for the purposes of this paper, we focus on exact estimation using straightforward methods. Section 3 describes a simple but complete Kriegspiel player, combining both state estimation and move

selection, and explains how self-play was used to generate the first database of Kriegspiel checkmate problems with observation histories.

Section 4 develops the basic AND–OR tree structure for solving Kriegspiel-like games, allowing for possibly-illegal moves. Section 5 defines two baseline algorithms—versions of depth-first search (DFS) and proof-number search (PNS)—for solving such trees, and then presents some basic improvements for these algorithms and analyzes their performance on our checkmate database.

Section 6 develops a new family of *incremental* search algorithms that treat uncertainty as a new search dimension in addition to depth and breadth, incrementally proving belief states by adding a single constituent “physical state” to a solution at each step. This leads to a further speedup of one or more orders of magnitude. Finally, Section 7 shows how state estimation and incremental checkmate search may be interleaved.

2 State estimation

If we are to find a guaranteed win for White, state estimation must identify the (logical) *belief state*—the set of all *physical states* (configurations for White and Black pieces) that are consistent with White’s history of moves and observations. A naive algorithm for exact state estimation looks like this:

- The initial belief state is a singleton, because Black’s pieces start the game in their normal positions.
- For each White move attempt, apply the move to every physical state and remove those states that are inconsistent with the subsequent percept.
- For each Black turn,
 - For each sequence of k “Illegal” percepts for the unobserved Black move attempts, remove any physical state for which there are fewer than k distinct illegal moves.
 - Replace each remaining physical state by the set of updated states corresponding to all Black moves that are legal and yield the given percept when made in that state.
 - Remove duplicate states using a transposition table.

In Kriegspiel, the belief state can grow very large— 10^{10} states or more—so this algorithm is not always practical. We have found, however, that with certain aggressive styles of play, the belief state remains no larger than a few thousand states throughout the game.

The naive algorithm given above can be viewed as a breadth-first expansion of a physical-state tree with branching at Black’s moves and pruning by percepts. An alternative method, which we adopt, performs a depth-first search of the tree instead. This has the advantage of generating a stream of consistent current states with only a moderate amount of effort per state. Furthermore, if the set of states found so far does not admit a checkmate, then the whole belief state does not admit a checkmate and both state estimation and move selection can be terminated early (see Section 7).

A randomized depth-first search can generate a randomly selected sample of consistent states that can be used for approximate decision making. We explore approximate state estimation in a subsequent paper; for now, we assume that the exact belief state is available to the checkmate algorithm.

3 A Kriegspiel checkmate database

In order to evaluate checkmate-finding algorithms for Kriegspiel, we require a database of test positions. Prior to this work, no such database existed.³ Our first database consists of White's move and percept histories for 1000 Kriegspiel games, up to the point where a 3-ply checkmate might exist for White.⁴ 500 of these are actual mate instances; the other 500 are *near-miss* instances, which "almost" admit a guaranteed checkmate within 3 ply. For each near-miss instance, there is a checkmate plan that works in at least half of the possible physical states, but not in all of them.

This database was created by analyzing games between two different Kriegspiel programs. The first program, playing White, performs exact state estimation and makes a complex static approximation to 2-ply lookahead; it plays well but can be defeated easily by a skilled human.⁵ The second program, playing Black, is much weaker: it computes a limited subset of its true belief state and attempts moves that are most likely to be captures and checks first. Whenever White's belief state has 100 Black positions or fewer, we determine if the belief state describes a mate or near-miss instance. If so, the move and percept history for the game-in-progress is saved. Games in which White's belief state grows above 10,000 positions are excluded. With these two programs, White's belief state generally remains fairly small, and about half the games played result in problem instances.⁶

Obviously, the checkmate problems we generate by this method have belief states that never exceed 10,000 physical states throughout the move history and have at most 100 physical states in the final position (the average is 11). Furthermore, the solution is at most 3-ply, but may include branches with illegal move attempts in addition to 3 actual moves.

By simply re-analyzing our 3-ply near-miss problems at 5-ply, we have also constructed a more difficult database of 258 5-ply mate instances and 242 5-ply near-miss instances. Both databases are available at our website (URL in Section 1).

As better state estimation, search, and evaluation methods are developed, it will be possible to construct more difficult problems that better reflect the kinds of positions reached in expert play. Nonetheless, our problems are far from trivial; for example, we will see that on 97% of the 5-ply near-miss instances, basic depth-first search requires more than 2000 CPU seconds to determine that no mate exists (within 5 ply).

4 Guaranteed Kriegspiel checkmates

Thanks to Theorem 1, our search problem involves a tree whose nodes correspond to White's belief states. Figure 1 shows a simple example: a miniature (4x4) 3-ply Kriegspiel checkmate. In the root belief-state node (1) there are three

³The Internet Chess Club has a database of several thousand Kriegspiel games, but guaranteed wins cannot be identified because the database omits the history of attempted moves.

⁴In fact, we find deeper mates through the simple expedient of classifying leaf nodes as wins if Black is checkmated or if White has a known material win, as defined above.

⁵For ordinary play, we combine this move selection algorithm with an online version of our approximate depth-first state estimation algorithm.

⁶To achieve this high efficiency, we add extra illegal move attempts to White's move history as "hints"; without these hints, fewer games satisfy our belief-state size criteria.

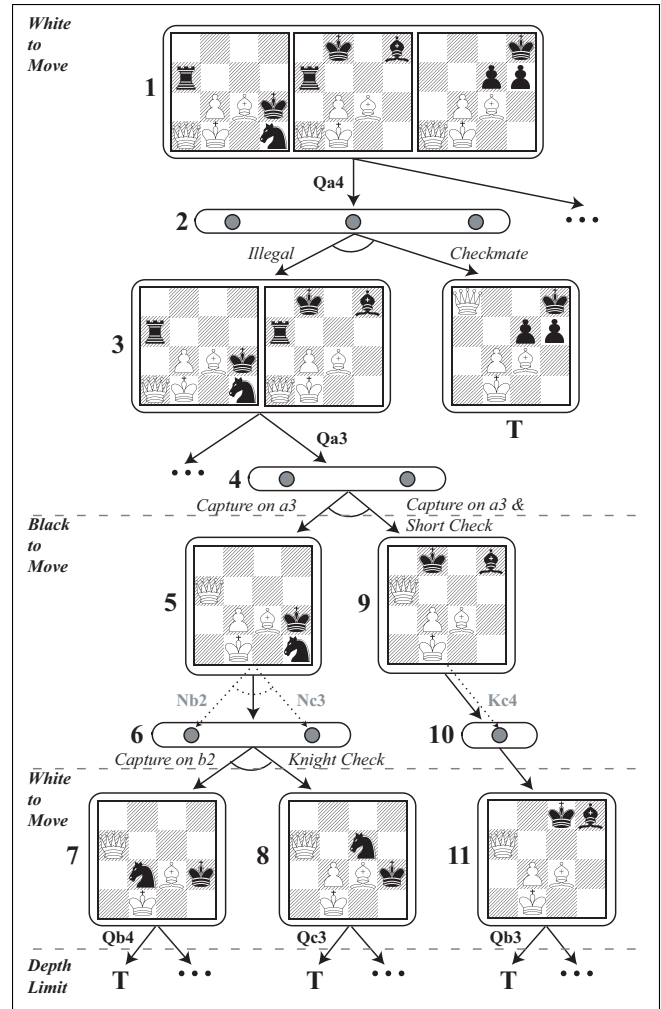


Figure 1: A minimal AND-OR proof tree for a 4x4 Kriegspiel 3-ply checkmate problem. The grayed moves in the Black-to-Move section are hidden from White.

possible physical states, which differ in the locations and types of Black's pieces (White will always know the number of remaining Black pieces). The figure depicts a minimal proof tree for the problem instance, with other possible moves by White omitted; it describes the following strategy:

1. White attempts move Qa4 from belief state 1. If the right-most state (1.c) is the true state, White wins.
2. Otherwise, Qa4 was illegal and White now attempts move Qa3 from belief state 3.
 - (a) If the subsequent percept is "Capture on a3," Black has two legal moves: Nb2 and Nc3.
 - i. If Black makes Nb2, the referee announces "Capture on b2" and White mates with Qb4.
 - ii. If Black makes Nc3, the referee announces "Knight Check" and White mates with Qc3.
 - (b) If the subsequent percept is "Capture on a3 & Short (Diagonal) Check," Black has only one legal move: Kc4. White mates with Qb3.

In general, belief-state AND-OR trees consist of three types of nodes:

- OR-nodes: In Figure 1, OR-nodes appear in the *White*

to *Move* sections (e.g., nodes 1, 3, 7). An OR-node represents a choice between possible moves for White, and is proven iff at least one of its children is proven. Its children are AND-nodes, each containing the results of applying a single move in every possible physical state.

- **EXPAND-nodes:** EXPAND-nodes appear in the *Black to Move* sections, representing Black’s moves (e.g., nodes 5, 9). Since Black’s moves are invisible to White, each EXPAND-node has only a single child, an AND-node containing the union (eliminating duplicates) of the legal successors of its possible physical states. An EXPAND-node is proven iff its only child is proven.
- **AND-nodes:** AND-nodes are the thin nodes that appear at every other level in the tree (e.g., nodes 2, 4, 6). Physical states within AND-nodes are abbreviated as circles. An AND-node represents the arrival of a percept from the referee, and can be terminal or non-terminal:
 - If every physical state in an AND-node is a terminal win for White, the node is terminal with value *true*. If *any* physical state is a terminal draw or loss for White, the node is terminal with value *false*.
 - Otherwise, the AND-node is nonterminal, and has children that form a partition of its nonterminal physical states (percepts do not change the underlying physical states—see, e.g., nodes 7 and 11).

Thus, an AND-node is proven iff all of its belief-state-tree children *and* its terminal physical states are proven.

In Kriegspiel, the referee makes an announcement after each move attempt. Thus, Kriegspiel belief-state trees have AND-nodes at every other level. The intervening nodes alternate between EXPAND-nodes (Black moves) and sequences of OR-nodes (White move attempts).⁷ Because one turn for White may involve several move attempts, White’s entire turn has a worst-case branching factor equal to the factorial of the number of possible moves.

5 Searching belief-state AND–OR trees

This section describes two common algorithms—depth-first search and proof-number search—for searching belief-state AND–OR trees. Like other existing algorithms, both solve a belief-state tree as an ordinary AND–OR tree with black-box belief-state nodes. After introducing the algorithms, we evaluate their performance on our 5-ply checkmate database, with and without some basic improvements.

5.1 DFS and PNS

The pseudocode for **DFS** (depth-first search) is shown in Figure 2.⁸ **DFS** operates using the **EXPAND** method, which constructs and evaluates the children of a belief-state node (as described in Section 4); as an example, Figure 3 shows **EXPAND**’s OR-node instance. To use **DFS**, we simply initialize an OR-node with the root belief state and remaining depth, and pass it to **SOLVE-TOP**. In Figure 1, the numbers beside the nodes indicate an order in which **DFS** might expand them when searching the tree.

⁷Thanks to Theorem 1, illegal Black moves are not considered.

⁸The pseudocode we present in this paper was written for simplicity, and does not include modifications necessary for handling possibly-illegal moves. Our actual implementations are also more efficient (for instance, they construct only one child at a time at OR-nodes), and thus differ significantly from the pseudocode shown.

```

function SOLVE-TOP(b) returns true or false
  inputs: b, a belief-state node

  EXPAND(b)
  return SOLVE(b)



---


method SOLVE(b an OR-node) returns true or false
  while CHILDREN(b) is not empty do
    if SOLVE-TOP(FIRST(CHILDREN(b))) then return true
    POP(CHILDREN(b))
  return false



---


method SOLVE(b an EXPAND-node) returns true or false
  return SOLVE-TOP(CHILD(b))



---


method SOLVE(b an AND-node) returns true or false
  if TERMINAL(b) then return VALUE(b)
  while CHILDREN(b) is not empty do
    if not SOLVE-TOP(FIRST(CHILDREN(b))) then return false
    POP(CHILDREN(b))
  return true

```

Figure 2: The **DFS** algorithm.

```

method EXPAND(b an OR-node)
  for each m in MOVES(FIRST(STATES(b))) do
    b' ← a new AND-node with TERMINAL(b') = false,
      VALUE(b') = true, DEPTH(b') = DEPTH(b),
      CHILDREN(b') = an empty list, and
      STATES(b') = MAP(SUCCESSOR(*, m), STATES(b))
    for each s in STATES(b') do
      if s is a win for White then remove s from STATES(b')
      else if s is terminal or DEPTH(b) = 1 then
        b' ← false; break
    if b' ≠ false then
      PUSH(b', CHILDREN(b))
      if STATES(b') is empty then
        TERMINAL(b') ← true; break

```

Figure 3: The OR-node instance of the **EXPAND** method, which constructs and evaluates the children of *b*.

PNS (proof-number search) is a best-first search algorithm for AND–OR trees, and is commonly believed to be superior to **DFS**. At each step **PNS** expands a “most-proving” node, which can make the largest contribution to proving or disproving the entire tree. A most-proving node is defined as any node that is a member of *both* a *minimal proof set* and a *minimal disproof set* of the tree, where a minimal proof/disproof set is a minimal-cardinality set of unexpanded nodes that, if proved/disproved, would be sufficient to prove/disprove the root. Every tree has at least one most-proving node; if there are multiple most-proving nodes, the **PNS** algorithm chooses one arbitrarily [Allis, 1994].⁹

5.2 Analysis and Improvements

Figure 4 shows the solving ability of our search algorithms on the 500 problems in our 5-ply database (for readability, we show only a subset of the algorithms tested). We will

⁹We alter the initialization of **PNS**’s tree to reflect the fact that wins occur only after White moves, but do not attempt to take the depth limit [Allis, 1994] or the amount of uncertainty [Sakuta, 2001] into account.

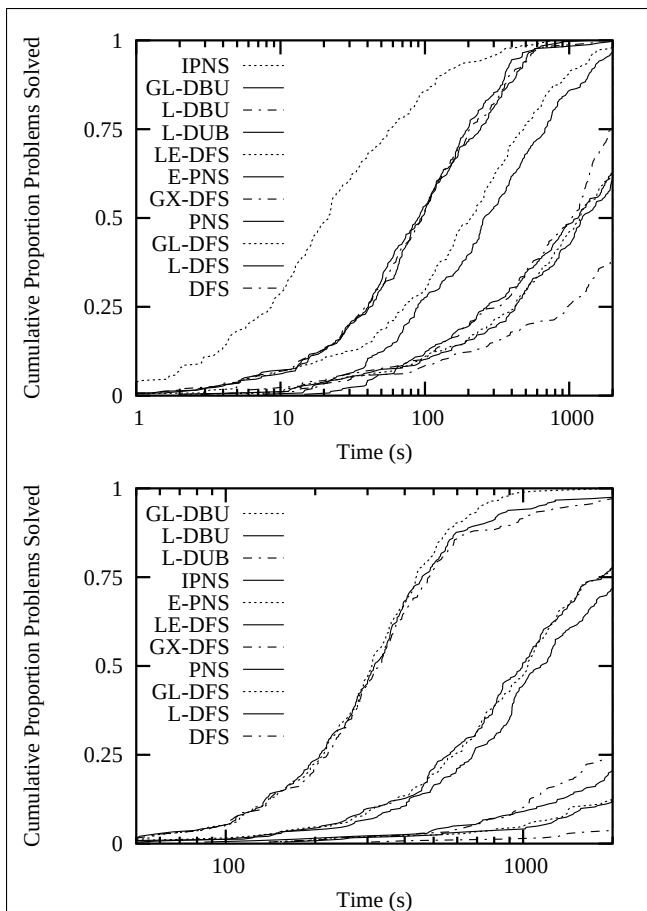


Figure 4: Performance of search algorithms on our 5-ply Kriegspiel checkmate database. Top: mate instances; Bottom: near-miss instances. The y -axes show the fraction of problems solvable within a given amount of CPU time (in Lisp, on a 550 MHz machine). The algorithms are ranked in decreasing order of efficiency.

introduce the **DBU**, **DUB**, and **IPNS** algorithms later, in Section 6. Performance on our 3-ply database (not shown) is qualitatively similar, but does not allow for accurate discrimination between our improved algorithms.

Basic **DFS** is by far the slowest of the algorithms tested, primarily because of the factorial branching factor for White (which subsequent algorithms avoid, to a large extent); basic **PNS** is much faster. Notice that the near-miss instances are generally more difficult to solve than the mate instances.

Heuristic ordering

When searching a belief-state AND-OR tree using a black-box algorithm such as **DFS**, there are two possible opportunities for heuristic ordering: White moves at OR-nodes, and percepts at AND-nodes. In this paper we focus on the underlying search algorithms; we do not investigate heuristic orderings for the White moves, and test only a simple but effective ordering for the percepts.

At AND-nodes, the *legal children* (children in which the last move was legal) are generally much cheaper for **DFS** to explore than the *illegal child*, since they have lower remaining depth. This suggests a simple heuristic: investigate the legal children first. As shown in Figure 4, **L-DFS** (Legal-first DFS) is considerably faster than **DFS**. On the other hand, **L-**

PNS (not shown) performs almost identically to **PNS** (which naturally allocates its efforts efficiently).

Future work may investigate the effects of ordering the White moves (e.g., information-gathering and likely checking moves first) and the legal percepts (e.g., checks and captures first for Black and last for White).

Pruning

Because a proof of guaranteed checkmate is a single branching plan that must succeed in every physical state of a belief state, we can make the following observation:

Theorem 2 *If a belief state does not admit a guaranteed checkmate, no superset of that belief state admits a guaranteed checkmate.*

A straightforward implementation of the **EXPAND** method (e.g., Figure 3) constructs all elements of a belief state before evaluating any of them. Theorem 2 suggests a more efficient strategy: evaluate each physical state as soon as it is constructed. If a terminal physical state with value *false* (or a nonterminal physical state at the depth limit) is found, the construction of the belief state can be halted early. In the best case, this reduces the effective search depth by one level (since only a single element of each belief state at the depth limit will be constructed). As shown in Figure 4 (indicated by E- for “early termination”), this simple idea is the most effective of the improvements we consider in this section.

Theorem 2 also suggests another pruning, which is specific to try-until-feasible trees. Consider the situation in which White is in belief state b , attempts a possibly-legal move, and is told that the move is illegal. White’s new belief state $b' \subset b$. Theorem 2 implies that if b' does not admit a guaranteed checkmate, then neither does b . In other words, when the illegal child of an AND-node is disproved, this is sufficient to disprove the AND-node’s parent OR-node as well. For example, if node 3 in Figure 1 were disproved, that would show not only that trying Qa4 first fails to ensure checkmate, but also that *no other White move from node 1 gives checkmate*. Clearly, this is a useful pruning rule.

We call this pruning *greedy*, since when combined with the legal-first heuristic it allows White’s turns to be solved without backtracking, by adding moves to the plan iff they lead to checkmate when legal. Because a move plan cannot include repetitions, a greedy algorithm such as **GL-DFS** (Greedy Legal-first DFS) has a worst-case branching factor per White’s turn that is only quadratic in the number of possible moves. However, Figure 4 shows that **GL-DFS** only slightly outperforms **L-DFS**. This is because the pruning only applies when there are moves that lead to checkmate if legal but not if illegal.

Perhaps surprisingly, our experiments show that a **G-DFS** algorithm performs better when it tries the *illegal child* first instead (even though the resulting algorithm is *not* actually greedy); this algorithm, shown as **GX-DFS** in Figure 4, outperforms even **PNS**. The power of **GX-DFS** stems from its ability to test a subset of its belief state using possibly-legal moves, and terminate early if it disproves the subset.

We did not implement a “**G-PNS**” algorithm, because the greedy pruning could force **PNS** to choose between the goals of proving and disproving the root (it always does *both simultaneously*). Future work may explore this issue further.

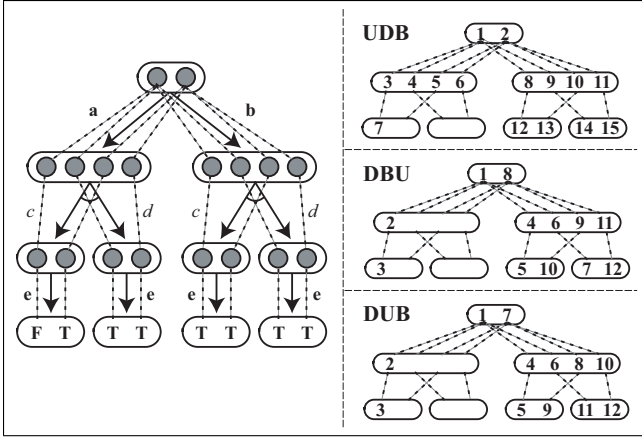


Figure 5: Left: a simple belief-state tree for a planning domain with nondeterministic transitions. **a**, **b**, and **e** are moves; **c** and **d** are percepts. Right: for each incremental algorithm, the order in which it would expand the nonterminal physical states in the tree.

6 Incremental belief-state AND-OR search

As we saw in the previous section, early termination via interleaved belief-state construction and evaluation can lead to large improvements in performance. This section develops this *incremental* idea into a novel framework for belief-state AND-OR tree search, which treats uncertainty as a new search dimension in addition to depth and breadth. After introducing this framework, we present results and theoretical analysis for our new algorithms.

6.1 Introduction

Ordinary AND-OR trees have two dimensions: depth and breadth. This leads to two “directional” search algorithms, depth- and breadth-first search, as well as numerous “best-first” algorithms (e.g., **PNS**). In addition to depth and breadth, belief-state AND-OR trees have uncertainty over physical states. By recognizing uncertainty as a new possible dimension for search, we can construct a new class of directional belief-state AND-OR search algorithms, as well as new best-first algorithms that balance all three factors efficiently.

In this paper, of the possible incremental directional algorithms, we consider only the three that put depth before breadth, which we will call **UDB**, **DBU**, and **DUB**. Figure 5 shows a simple belief-state tree for a domain with nondeterministic transitions, as well as the order that each of these algorithms would expand the physical states in the tree. The first algorithm, **UDB** (uncertainty-then-depth-then-breadth), is in fact just the **E-DFS** algorithm discussed in Section 5. In the figure, the difference between **UDB** and the other new algorithms should be immediately apparent; whereas **UDB** expands all physical states at a node before moving to the next node, the other algorithms begin by exploring the first *physical-state tree* in a depth-first manner. Thus, unlike existing algorithms, **DBU** and **DUB** can construct *minimal disproofs* that consider only a single element of each belief state.

In the tree, the difference between **DBU** and **DUB** first arises when selecting the seventh node for expansion. After establishing a proof on a single physical-state branch (i.e., non-branching path from the root to a leaf), **DBU** gives precedence to verifying the proof on the current *physical-state tree*, whereas **DUB** gives precedence to verifying it on the current *belief-state branch*. Thus, all three algorithms con-

```

function SOLVE-TOP(b) returns true or false
inputs: b, a belief-state node

while STATES(b) is not empty do
  INCREMENTAL-EXPAND(b, POP(STATES(b)))
  if not SOLVE(b) then return false
return true



---


method SOLVE(b an AND-node) returns true or false
if TERMINAL(b) then return VALUE(b)
return ( $\forall b' \in \text{CHILDREN}(b)$ ) SOLVE-TOP(b')

```

Figure 6: The **DBU** algorithm (which builds upon **DFS**).

struct proofs by “looking inside” the belief state; they differ in that **UDB** incrementally constructs belief-state *nodes*, whereas **DUB** incrementally constructs *branches* and **DBU** incrementally constructs entire *proof trees*.

At each point, **DBU** expands the deepest unexpanded physical state within the current proof tree. **DUB** does the same, except limited to a *single belief-state branch* at a time. Thus, in a pure OR-tree with no percept branching, **DUB** and **DBU** act identically. This brings us to an important point: the breadth that our **B** refers to is only the breadth of a proof, the AND-branching (percepts).

Whereas the algorithms differ significantly with respect to establishing disproofs, when exploring a proof tree such as the right branch of Figure 5, all three algorithms expand the *same* physical states, just in a different order. Since **UDB** and **DUB** both put breadth last, they explore the same sequence of belief-state branches, with different orderings for physical states within each branch. Likewise, **DUB** and **DBU** explore the same first physical-state branch.

In addition to these directional algorithms, we have implemented a best-first **IPNS** (incremental **PNS**) algorithm that operates on a single physical state at a time. This algorithm uses the above tree model, allowing AND-nodes to store unexpanded physical states. By simply redefining a most-proving node as a *physical state* that, if expanded, could contribute most to the proof/disproof of the entire tree, the proof-number idea naturally generalizes over uncertainty as well as depth and breadth. Among other things, this allows **IPNS** to naturally consider the relative ease of proving and disproving its belief-state nodes based on their sizes, an ability which other researchers have attempted to artificially introduce into a **PNS**-type algorithm [Sakuta, 2001].

6.2 Implementations

Our implementation of **DBU**, shown in Figure 6, uses a new **INCREMENTAL-EXPAND** method that expands a single physical state rather than an entire belief state at a time (its OR-node instance is shown in Figure 8, for comparison with Figure 3). When **DBU**’s **SOLVE-TOP** encounters uncertainty, it first constructs a proof for a single state, and then extends the proof to cover additional states one-at-a-time. To support such incremental proofs, **DFS**’s **SOLVE** instance for AND-nodes must also be modified to save proved children, rather than popping them; this allows **DBU** to continually refine a single proof tree that works in all physical states examined so far.

Our implementation of **DUB** uses two sets of recursive methods. The inner recursion is exactly that of **DBU**, ex-

```

function OUTER-TOP(b) returns true or false
  inputs: b, a belief-state node

  return (SOLVE-TOP(b) and OUTER(b))



---


method OUTER(b an OR-node) returns true or false
  loop do
    if OUTER(FIRST(CHILDREN(b))) then return true
    POP(CHILDREN(b))
    if not SOLVE(b) then return false



---


method OUTER(b an EXPAND-node) returns true or false
  return OUTER(CHILD(b))



---


method OUTER(b an AND-node) returns true or false
  if TERMINAL(b) then return VALUE(b)
  loop do
    if not OUTER(FIRST(CHILDREN(b))) then return false
    POP(CHILDREN(b)) /* percept branching here */
    if CHILDREN(b) is empty then return true
    if not SOLVE(b) then return false



---


method SOLVE(b an AND-node) returns true or false
  if TERMINAL(b) then return VALUE(b)
  return SOLVE-TOP(FIRST(CHILDREN(b))) /* not here */

```

Figure 7: The **DUB** algorithm (which builds upon **DBU**)

```

method INCREMENTAL-EXPAND(b an OR-node, s a state)
  if CHILDREN(b) is empty then /* create b's children */
    for each m in MOVES(s) do
      b' ← a new AND-node with TERMINAL(b') = true,
        VALUE(b') = true, DEPTH(b') = DEPTH(b),
        CHILDREN(b') = an empty list, MOVE(b') = m,
        and STATES(b') = an empty list
      PUSH(b', CHILDREN(b))
    for each b' in CHILDREN(b) do /* integrate s's children */
      s' ← SUCCESSOR(s, MOVE(b'))
      if s' is terminal or DEPTH(b) = 1 then
        if s' is not terminal or s' is not a win for White then
          remove b' from CHILDREN(b)
        else PUSH(s', STATES(b')); TERMINAL(b') ← false

```

Figure 8: The OR-node instance of the INCREMENTAL-EXPAND method, which constructs and evaluates the children of *s*, integrating them into the children of *b* (which are also constructed if necessary).

cept that the SOLVE method for AND-nodes is modified to test only the first percept encountered (rather than all possible percepts); one might call this modified recursion simply **DU**. It either returns *false*, indicating a certain disproof, or *true*, representing a partial proof of a single belief-state-tree branch. The outer recursion, consisting of OUTER-TOP and OUTER, uses the inner recursion to construct a partial proof and then verify this proof on other percepts (deepest-first).

When implementing **DUB** or **DBU** in a try-until-feasible domain, a new issue arises: potential White moves that are *always illegal* are useless, but inflate the branching factor substantially; thus, it is crucial to avoid them during search. This is trivial for an uncertainty-first algorithm, since always-illegal moves can be filtered out during move generation. However, an incremental algorithm cannot use this method, because in general only a single physical state will be avail-

able when constructing a belief-state node. To avoid the large penalty associated with always-illegal moves, our actual implementations of **DUB** and **DBU** use the legal-first heuristic and skip the move in question (saving it for a later attempt) if it is not legal in any states examined so far.

With incremental search, there are also new opportunities for heuristic orderings that we have not yet investigated. For one, the physical states within a belief state can be ordered (e.g., best for Black first). One might also consider dynamic move orderings, using physical-state and/or belief-state transposition tables to cache proving moves; this could be especially effective in combination with iterative deepening.

6.3 Results

In Figure 4, we see that the directional incremental algorithms have significantly higher solving ability than their non-incremental counterparts. The true depth-first algorithms (**L-DUB** and **L-DBU**) perform at a similar level, outpacing **L-UDB** (**LE-DFS**) by a large margin. Again, greedy pruning has a small but significant effect: **GL-DBU** has the highest solving ability of the algorithms tested, solving 499 of 500 5-ply problems within the 2000-second time limit.¹⁰

In the figure, we see that **IPNS** is by far the most effective of our algorithms in solving the mate instances, but falls behind the true depth-first algorithms on the near-miss instances. This discrepancy can be explained by the depth limit, which strongly violates a basic assumption of **IPNS**: that the expected amount of work to disprove a physical state is constant throughout the tree. Thus, we expect that the discrepancy would disappear after adapting **IPNS** to the depth limit, or when searching without one.

6.4 Analysis

In this section, we conduct a brief analysis of the time and space complexity of our new algorithms. No directional algorithm is best in general; for specific classes of belief-state trees, however, clear differences do arise between the algorithms. In the following analysis, we focus on disproofs (since the algorithms generate the same trees for proofs), and ignore illegal moves and transpositions.

Recall that in any tree with all terminal leaves at the depth limit, **DFS** dominates **BFS** in the sense that for every fixed branch ordering, the set of nodes expanded by **DFS** will be a subset of the set of nodes expanded by **BFS**. We can make an analogous claim comparing the operation of **DUB** and **UDB**:

Theorem 3 *In a tree with all false leaves at the depth limit, for any fixed branch ordering, the set of physical states expanded by **DUB** will be a subset of the set of physical states expanded by **UDB**.*

In this class of trees, **UDB** and **DUB** visit the same set of belief-state nodes with the same order of first visit. However, **DUB** does depth-first rather than uncertainty-first searches of each belief-state-tree branch, allowing it to find the *false* leaves faster. Theorem 3 nearly holds for our problem database, because shallow *false* leaves arise only from stalemates and Black checkmates, which are relatively rare in the positions we create.

¹⁰Incidentally, unlike any of our other algorithms, when a move in its current plan is disproved, **GL-DBU** can salvage the remainder of the plan.

Using a simple tree model, we can also approximate the best-case speedup and worst-case memory requirements for our new algorithms. Consider a belief-state tree rooted at an OR-node of size u_0 , with depth d and fixed branching factors m_W , m_B , p_W , and p_B for the White and Black moves and percepts. In this tree, examine an arbitrary EXPAND-node 2-ply from the depth limit with size u' , and define $u = u' * m_B$. If the belief-state tree has no terminal nodes, then the following table shows how many physical states each directional algorithm must construct to disprove the EXPAND-node (not including elements of the EXPAND-node itself):

DFS	$u + (u/p_B) * m_W$
UDB	$u + m_W$
DUB & DBU	$1 + m_W$

Since a majority of the tree's physical states will be located within 2-ply of the depth limit, we can approximate the overall performance of our search algorithms by the number of physical states they construct within its deepest 2-ply. Furthermore, because all four algorithms visit the same set of belief-state nodes in trees without terminal nodes, by setting u' to the *average* belief-state size of visited EXPAND-nodes 2-ply from the depth limit, we can interpret the values in the above table as approximately proportional to *run times*. Thus, in the best case, **DUB** and **DBU** are faster by roughly a factor of the average belief-state size in the tree. This is consistent with our observed speedup: in our 5-ply database, the average value of u (as defined above) is approximately 60.

Under the above tree model, with the additional stipulation that physical states be evenly distributed among percepts, the worst-case asymptotic memory requirements for efficient implementations of the algorithms are as follows:

DFS, UDB, & DUB	$O(u_0 * p_B \sum_{i=0}^{\lfloor d/2 \rfloor} (\frac{m_B}{p_B * p_W})^i)$
DBU	$O(u_0 * \sum_{i=0}^{\lfloor d/2 \rfloor} (m_B)^i)$
PNS & IPNS	$O(u_0 * \sum_{i=0}^{\lfloor d/2 \rfloor} (m_B m_W)^i)$

UDB and **DUB** store only a proving branch plus physical states for other possible percepts, whereas **DBU** must store a proof tree and **IPNS** must store the entire belief-state tree. Because of **IPNS**'s large memory requirements, one might attempt to construct a depth-first variant of the algorithm, analogous to recent work on ordinary **PNS** [Sakuta, 2001].

7 Interleaved state estimation and search

The depth-first method for state estimation described in Section 2 can be interleaved with the **DBU** checkmate-finding algorithm described in Section 6. As each new state is found by the state estimation algorithm, it is integrated into the current proof tree. This process continues until a disproof is found (early termination) or the entire belief state has been proven.

Computation times for “interleaved” vs. “sequential” methods (using **GL-DBU**) applied to each 3-ply database instance are shown in Figure 9. As expected, interleaving can provide substantial time savings on near-miss instances by eliminating the need for full state estimation, but has no effect on the solving of mate instances.

8 Conclusions and further work

We have proposed a new family of statewise-incremental solvers for belief-state AND-OR trees, and have shown them

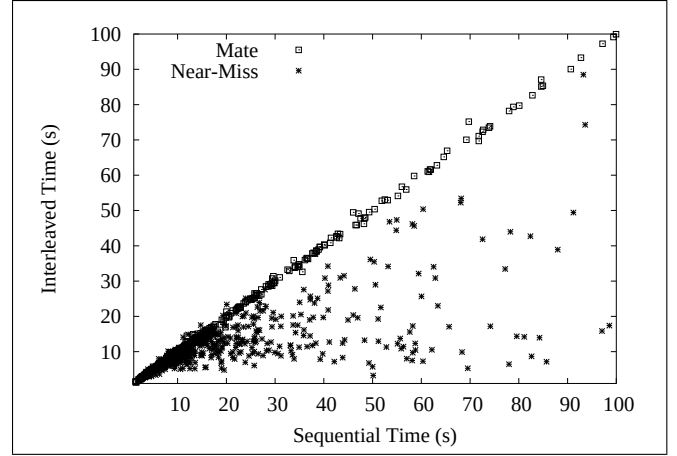


Figure 9: Results for interleaved state estimation and search.

to yield large performance improvements on a database of Kriegspiel checkmate problems. Future work will enhance our complete Kriegspiel player with belief-state transposition tables (as explored by Sakuta [2001]) and improved methods for approximate state estimation and nonterminal evaluation, as well as evaluate further applications of incremental belief-state search. In particular, we plan to investigate dynamic move orderings and iterative deepening, further analyze the combination of incremental search and approximate state estimation, and apply incremental search to existing methods for general play. One might also consider incrementalization of partially observable planners and of QBF solvers more generally.

References

- [Allis, 1994] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, 1994.
- [Bertoli et al., 2001] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI*, 2001.
- [Bolognesi and Ciancarini, 2003] A. Bolognesi and P. Ciancarini. Computer Programming of Kriegspiel Endings: The Case of KR vs. K. In *Advances in Computer Games 10*, 2003.
- [Bolognesi and Ciancarini, 2004] A. Bolognesi and P. Ciancarini. Searching over Metapositions in Kriegspiel. In *Computers and Games 2004*. Springer-Verlag, 2004.
- [Ciancarini et al., 1997] P. Ciancarini, F. DallaLibera, and F. Maran. Decision Making under Uncertainty: A Rational Approach to Kriegspiel. In *Advances in Computer Chess 8*, 1997.
- [Ferguson, 1992] T. Ferguson. Mate with Bishop and Knight in Kriegspiel. *Theoretical Computer Science*, 96:389–403, 1992.
- [Ferguson, 1995] T. Ferguson. Mate with the Two Bishops in Kriegspiel. Technical report, UCLA, 1995.
- [Ginsberg, 1999] M. L. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *IJCAI*, 1999.
- [Koller and Pfeffer, 1997] D. Koller and A. Pfeffer. Representations and solutions for game-theoretic problems. *Artificial Intelligence*, 94:167–215, 1997.
- [Parker et al., 2005] A. Parker, D. Nau, and V. S. Subrahmanian. Game-tree search with combinatorially large belief states. In *IJCAI*, 2005. (In press).
- [Russell and Norvig, 2003] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2003.
- [Sakuta, 2001] M. Sakuta. *Deterministic Solving of Problems with Uncertainty*. PhD thesis, Shizuoka University, 2001.

Bounded Search and Symbolic Inference for Constraint Optimization

Martin Sachenbacher and Brian C. Williams

Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Cambridge, MA 02139
{sachenba,williams}@mit.edu

Abstract

Constraint optimization underlies many problems in AI. We present a novel algorithm for finite domain constraint optimization that generalizes branch-and-bound search by reasoning about sets of assignments rather than individual assignments. Because in many practical cases, sets of assignments can be represented implicitly and compactly using symbolic techniques such as decision diagrams, the set-based algorithm can compute bounds faster than explicitly searching over individual assignments, while memory explosion can be avoided by limiting the size of the sets. Varying the size of the sets yields a family of algorithms that includes known search and inference algorithms as special cases. Furthermore, experiments on random problems indicate that the approach can lead to significant performance improvements.

1 Introduction

Many problems in AI, such as planning, diagnosis, and autonomous control, can be formulated as finite domain constraint optimization problems [Schiex *et al.*, 1995]. Thus, the ability to solve large instances of optimization problems efficiently is key for tackling practical applications.

Depth-first branch-and-bound finds optimal solutions by searching through the space of possible assignments. To prune parts of the search tree, it computes a lower bound on the value of the current partial assignment, and compares it with the value of the best solution found so far as an upper bound. While branch-and-bound is memory-efficient, it can lead to impractical run-time, because the size of the search tree can grow exponentially as its depth increases.

An alternative approach is to infer optimal solutions by repeatedly combining constraints together. This approach is search-free and thus does not suffer from the run-time complexity of backtracking; however, its exponential memory requirements can render this approach infeasible as well.

In practical cases, however, constraints often exhibit a structure that can be exploited in order to reduce the memory requirements. Decomposition [Gottlob *et al.*, 2000] can exploit structural properties such as low induced width in order to break down the set of variables and constraints into smaller

subproblems. Likewise, symbolic encoding using decision diagrams [Bryant, 1986] can exploit regularities within sets of assignments (shared prefixes and postfixes) to collapse them into a much smaller representation. However, while these techniques can push the border on the size of the problems that can be handled, they still do not avoid the fundamental problem of memory explosion.

The idea presented in this paper is to extend branch-and-bound search to incorporate both decomposition and symbolic encoding. In particular, our algorithm simultaneously maintains sets of assignments (and thus sets of bounds) instead of single assignments. Because a set can, in many cases, be represented and manipulated efficiently using an implicit, symbolic representation (such as a decision diagram), the set-based search can compute bounds faster than by explicitly searching over the individual assignments, while memory explosion can be avoided by limiting the size of the sets. In our approach, similar to domain splitting, the size of the sets is controlled by partitions defined for the domain of each variable. By varying the granularity of the domain partitions, a family of tree-based algorithms is obtained that includes a recently introduced search algorithm called BTB (branch-and-bound on tree decompositions) [Terrioux and Jégou, 2003] and dynamic programming [Kask *et al.*, 2003] as limiting cases. We show that a trade-off exists between these two extremes, and thus for many practical cases, it is advantageous to pick an intermediate point along our spectrum.

The paper is organized as follows. After an introduction to the valued constraint satisfaction problem framework [Schiex *et al.*, 1995], we present a way to exploit structure in the constraints using tree decomposition and a data-structure for symbolic encoding known as algebraic decision diagrams (ADDs) [Bahar *et al.*, 1993]. We then describe the set-based extension of the branch-and-bound algorithm, and show how it generalizes existing algorithms. Finally, preliminary experiments on random problems illustrate the trade-off between search and inference and the benefit of a hybrid strategy.

2 Constraint Optimization Problems

Definition 1 (Constraint Optimization Problem) A constraint optimization problem (COP) consists of a tuple (X, D, F) with variables $X = \{x_1, \dots, x_n\}$, finite domains $D = \{d_1, \dots, d_n\}$, constraints $F = \{f_1, \dots, f_m\}$, and a valuation structure $(E, \leq, \oplus, \perp, \top)$. The constraints $f_j \in F$

are functions $f_j : d_1 \times \dots \times d_n \rightarrow E$ mapping assignments to X to values in E . E is totally ordered by $\leq$ with a minimum element $\perp \in E$ and a maximum element $\top \in E$, and $\oplus$ is an associative, commutative, and monotonic operation with identity element $\perp$ and absorbing element $\top$.

The set of valuations E expresses different levels of constraint violation, such that $\perp$ means satisfaction and $\top$ means unacceptable violation. The operation $\oplus$ is used to combine (aggregate) several valuations. A constraint is *hard*, if all its valuations are either $\perp$ or $\top$. For notational convenience, we denote by $x_i \leftarrow v$ both the assignment of value $v \in d_i$ to variable x_i , and also the hard constraint corresponding to this assignment (its valuation is $\perp$ if the value of x_i is v , and $\top$, otherwise). Likewise, we regard elements of E as values, but also as special cases of constraints (constant functions).

Definition 2 (Combination and Projection) Let $f, g \in F$ be two constraints. Let $t \in d_1 \times \dots \times d_n$, and let $t \downarrow_Y$ denote the restriction of t to a subset $Y \subseteq X$ of the variables. Then,

1. The combination of f and g , denoted $f \oplus g$, is the constraint that maps each t to the value $f(t) \oplus g(t)$;
2. The projection of f onto a set of variables Y , denoted $f \downarrow_Y$, is the constraint that maps each t to the value $\min\{f(t_1), f(t_2), \dots, f(t_k)\}$, where $t_1, t_2, \dots, t_k$ are all the assignments for which $t_i \downarrow_Y = t \downarrow_Y$.

Given a COP and a subset $Z \subseteq X$ of variables of interest, a *solution* is an assignment t with value $(\bigoplus_{j=1}^m f_j)(t) \downarrow_Z$. In particular, for $Z = \emptyset$, the solution is the value α^* of an assignment with minimum constraint violation, that is, $\alpha^* = (\bigoplus_{j=1}^m f_j) \downarrow_{\emptyset}$.

For example, the problem of diagnosing the full adder circuit in Fig. 1 can be framed as a COP with variables $X = \{u, v, w, y, a_1, a_2, e_1, e_2, o_1\}$. Variables u to y describe boolean signals and have domain $\{0, 1\}$. Variables a_1 to o_1 describe the mode of each gate, which can either be G (good), S1 (shorted to input 1), S2 (shorted to input 2), or U (unknown failure). The COP has five constraints $f_{a_1}, f_{a_2}, f_{e_1}, f_{e_2}, f_{o_1}$, one for each gate in the circuit. Each constraint expresses that if the gate is G then it correctly performs its boolean function; and if it is S1 (S2) then it is broken such that its output equals its first (second) input; and if it is U then it is broken in an unknown way and no assumption is made about its behavior. The valuation structure captures the likelihood of being in a mode, and is $([0, 1], \geq, \cdot, 1, 0)$ (with $\cdot$ being multiplication over the real numbers). We assume Or-gates have a .95 probability of being G, a .02 probability of being S1 (S2), and a .01 probability of being U; both And-gates and Xor-gates have a .975 probability of being G, a .01 probability of being S1 (S2), and a .005 probability of being U. The value of the best solution then corresponds to the most likely fault in the circuit. For the example, α^* is .018, corresponding to a stuck-at-first-input (S1) fault of Or-gate 1.

We introduce four more operators that we will use later to compare constraints and to turn them into hard constraints. The *minimum* operation, denoted $\min(f, g)$, returns the constraint whose valuation is the minimum of $f(t)$ and $g(t)$:

$$\min(f, g)(t) = \begin{cases} f(t) & \text{if } f(t) < g(t) \\ g(t) & \text{otherwise} \end{cases}$$

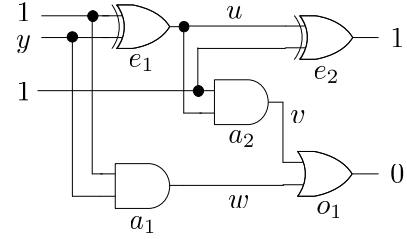


Figure 1: Full adder circuit consisting of two And gates, one Or gate, and two Xor gates. Input and output values are observed as indicated.

The *sinking* operation, denoted $\text{sink}(f, g)$ returns the constraint that forbids t if $f(t) \geq g(t)$:

$$\text{sink}(f, g)(t) = \begin{cases} f(t) & \text{if } f(t) < g(t) \\ \top & \text{otherwise} \end{cases}$$

The *lifting* operation, denoted $\text{lift}(f)$, turns a constraint into a hard constraint that allows t if $f(t) < \top$. Finally, the *complement* of a hard constraint f , denoted $\text{compl}(f)$, is the constraint whose valuation is $\top$ if $f(t) = \perp$, and $\perp$, otherwise.

3 Structure in Constraints

Our approach is based on exploiting independence properties of the constraint functions F , such that they can be represented more compactly (abstractly) than explicitly listing all assignments to X . In this section, we characterize these properties, which are often present in practical problems.

To start with, in many situations the valuation of a constraint will depend only on a subset of the variables. For instance, for the constraint f_{a_1} , its valuation depends only on a_1 , w , and y . Formally, the *support* of a constraint f is the subset of variables that it depends upon:

Definition 3 (Support) The support of a constraint f , denoted $\text{sup}(f)$, is the variable set $\{x_i \in X \mid \exists v_1, v_2 \in d_i \text{ s.t. } (f \oplus (x_i \leftarrow v_1)) \downarrow_{X \setminus \{x_i\}} \neq f \oplus (x_i \leftarrow v_2)) \downarrow_{X \setminus \{x_i\}}\}$.

For the example, $\text{sup}(f_{a_1}) = \{a_1, w, y\}$, $\text{sup}(f_{a_2}) = \{a_2, u, v\}$, $\text{sup}(f_{e_1}) = \{e_1, u, y\}$, $\text{sup}(f_{e_2}) = \{e_2, u\}$, and $\text{sup}(f_{o_1}) = \{o_1, v, w\}$. The support structure of a constraint problem can be abstractly represented through a hypergraph H that associates a node with each variable x_i , and a hyperedge with the variables $\text{sup}(f_j)$ of each constraint f_j . Fig. 3 shows the hypergraph for the example.

While the notion of support captures the independence of a function's value from variables outside the support, there can still exist independence within the support. For instance, in the constraint f_{o_1} , if $o_1 = S1$ and $v = 0$, then the value is .02 regardless of the value of w ; if $o_1 = U$, then the value is .01 regardless of the values for w and y , etc. More generally, a property that we call *weak support* can be exploited: if the relationship between assignments and a function's value can be described more compactly than explicitly listing all the assignments to the support, then it is more efficient to solve problems on that symbolic level.

3.1 Symbolic Encoding using Decision Diagrams

In the following, we present a way to recognize support and weak support of functions, namely through symbolic encoding in the form of decision diagrams.

A decision diagram represents a function over boolean variables to a set of values. Binary decision diagrams (BDDs) [Bryant, 1986] represent functions with values 0 or 1; algebraic decision diagrams (ADDs) [Bahar *et al.*, 1993] represent functions to any set of values. A decision diagram is a rooted, directed, acyclic graph, where each internal node corresponds to a boolean variable, and each leaf node corresponds to a value of the function. Internal nodes have two children n_1, n_2 , and are recursively interpreted as the function $f = \text{if } x_i \text{ then } f_1 \text{ else } f_2$, where x_i is the boolean variable corresponding to the node, and f_1 and f_2 interpret the sub-diagrams with root nodes n_1 and n_2 , respectively.

The power of decision diagrams derives from their reduction rules and canonicity of representation. A decision diagram can be ordered by imposing a variable ordering $x_1 \prec x_2 \prec \dots \prec x_n$, such that for all paths from the root to the leaves, the sequence of variables encountered obeys the ordering. Ordered decision diagrams can be reduced by iteratively applying two graph reduction rules, which collapse assignments by sharing common prefixes and postfixes (to share a common postfix, the function value must be the same): the node *deletion rule* eliminates nodes from the diagram whose children are equal ($n_1 = n_2$), and the node *sharing rule* eliminates one of two nodes that are root nodes of isomorphic sub-diagrams. A reduced, ordered decision diagram is a canonical representation of its function [Bryant, 1986] and contains only variables from the support of the function. It is easy to extend the technique to non-binary variables by mapping each non-binary variable x_i to a block of $\lceil \log_2 |D_i| \rceil$ boolean variables that encode the domain values logarithmically. Figure 2 shows a reduced, ordered ADD representing the function f_{o1} . Operations on functions, such as projection and combination, can be directly performed on this representation. The complexity of the operations depends on the size of the diagram (number of nodes and arcs), rather than on the number of possible assignments; due to the sharing of common substructures, the number of nodes and arcs can be orders of magnitude smaller than the number of possible assignments. While no compaction is achieved in the worst case, for certain types of constraints it can be shown that the size of the decision diagram grows only logarithmically with the number of assignments [Bryant, 1986].

4 Set-based Branch-and-Bound with Tree Decompositions

In this section, we describe how the independence properties of functions described in the previous section (support, weak support) can be exploited in the framework of branch-and-bound search. We describe an algorithm that uses a tree decomposition of the hypergraph H to exploit the support of functions, and set-based search to exploit the weak support of functions. Thus, the algorithm benefits from compact representations of the functions, while memory explosion is avoided through depth-first search.

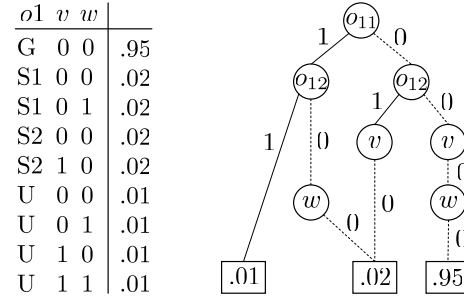


Figure 2: Constraint f_{o1} for the example in Fig. 1 and its ADD, using two binary variables o_{11}, o_{12} to encode o_1 , and variable ordering $o_{11} \prec o_{12} \prec v \prec w$. Assignments and paths with value 0 are not shown.

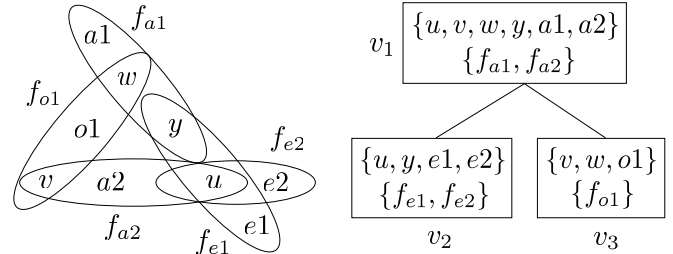


Figure 3: Hypergraph (left) and a tree decomposition (right) for the example in Fig. 1. The tree shows the labels χ and λ for each node.

4.1 Tree Decomposition

Tree decomposition [Gottlob *et al.*, 2000; Kask *et al.*, 2003] is a way to exploit structural properties of H to decompose the original problem into independent subproblems (“clusters”):

Definition 4 (Tree Decomposition) A tree decomposition for a problem (X, D, F) is a triple (T, χ, λ) , where $T = (V, E)$ is a rooted tree, and χ, λ are labeling functions that associate with each node (cluster) $v_i \in V$ two sets $\chi(v_i) \subseteq X$ and $\lambda(v_i) \subseteq F$, such that

1. For each $f_j \in F$, there exists exactly one v_i such that $f_j \in \lambda(v_i)$. For this v_i , $\text{var}(f_j) \subseteq \chi(v_i)$ (covering condition);
2. For each $x_i \in X$, the set $\{v_j \in V \mid x_i \in \chi(v_j)\}$ of vertices labeled with x_i induces a connected subtree of T (connectedness condition).

In addition, we demand that the constraints appear as close to the root of the tree as possible, that is,

3. If $\text{var}(f_j) \subseteq \chi(v_i)$ and $\text{var}(f_j) \not\subseteq \chi(v_k)$ with v_k the parent of v_i , then $f_j \in \lambda(v_i)$.

Figure 3 shows a tree decomposition for the example. The separator of a node, denoted $\text{sep}(v_i)$, is the set of variables that v_i shares with its parent node v_j : $\text{sep}(v_i) = \chi(v_i) \cap \chi(v_j)$. For convenience, we define $\text{sep}(v_{\text{root}}) = \emptyset$. Intuitively, $\text{sep}(v_i)$ is the set of variables that connects the subproblem rooted at v_i with the rest of the problem:

Definition 5 (Subproblem) For a COP and a tree decomposition (T, χ, λ) , the subproblem rooted at v_i is the COP that consists of the constraints and variables in v_i and any descendant v_k of v_i in T , with variables of interest $\text{sep}(v_i)$.

The subproblem rooted at v_{root} is identical to the problem of finding α^* for the original COP. The benefit of a tree decomposition is that each subproblem needs to be solved only once (possibly involving re-using its solutions); the optimal solutions can be obtained from optimal solutions to the subproblems using dynamic programming. Thus, the complexity of constraint solving is reduced to being exponential in the size of the largest cluster only.

In order to exploit the decomposition during search, the variables must be assigned in an order that is compatible with the tree, namely by first assigning the variables in a cluster before assigning the variables in the rest of the subproblems rooted in the cluster. This is called a *compatible order* in [Jégou and Terrioux, 2003]. In [Terrioux and Jégou, 2003], Jégou and Terrioux present an algorithm called BTB (backtracking with tree decompositions) that exploits tree decompositions in branch-and-bound search. BTB assigns variables along a compatible order, beginning with the variables in $\chi(v_{\text{root}})$. Inside a cluster v_i , it proceeds like classical branch-and-bound, taking into account only the constraints $\lambda(v_i)$ of this cluster. Once all variables in the cluster have been assigned, BTB considers its children (if there are any). Assume v_j is a child of v_i . BTB first checks if the restriction of the current assignment to the variables in $\text{sep}(v_j)$ has previously been computed as a solution to the subproblem rooted at v_j . If so, the value of this solution (called a “good”) is retrieved and combined with the value of the current assignment, thus preventing BTB from solving the same subproblem again (called a “forward jump” in the search). Otherwise, BTB solves the subproblem rooted at v_j for the current assignment to $\text{sep}(v_j)$ and the current upper bound, and records the solution as a new good. Its value is combined with the value of the current assignment, and if the result is below the upper bound, BTB proceeds with the next child of v_i .

4.2 Set-based Search

In the following, we generalize BTB from single assignments (and thus single bounds) to sets of assignment (and thus sets of bounds), in order to exploit symbolic representations of functions.

The method that we use to extend the search from single assignments to sets of assignments is to replace the step of assigning a value to a variable by the more general step of restricting a variable to a subset of its domain. This generalization is similar to domain splitting; however, whereas domain splitting might further split up the subsets at subsequent levels of the search tree, we consider the case where each variable occurs only once in each path of the search tree. That is, we assume that for each variable x_i , a static, predefined partition of its domain into subsets is given:

Definition 6 (Domain Partition) A partition of a finite domain d_i is a set P_i of disjoint subsets of d_i whose union is d_i , that is, $p_j \cap p_k = \emptyset$ for $p_j, p_k \in P_i$, $j \neq k$, and $\bigcup_{p \in P_i} p = d_i$.

There are two limiting cases: partitions consisting of singleton sets, that is, $|P_i| = |d_i|$, and partitions consisting of a single set containing all domain values, that is, $|P_i| = 1$. Again for notational convenience, we denote by $x_i \in p$ the restriction of a variable x_i to the values in a partition element p , and a constraint over variable x_i (its valuation is $\perp$ if the value of x_i is in p , and $\top$, otherwise).

The set-based algorithm proceeds by assigning partition elements p to variables x_i , and computing lower bounds by combining the constraints all of whose variables have been assigned. Since a partition element can contain more than one domain value, the result is in general a function (set of assignments) rather than a single assignment. Thus, we need to generalize the basic test of the branch-and-bound algorithm – comparing a lower bound with an upper bound – to comparing two functions:

Proposition 1 Given a COP with variables of interest $Z \subseteq X$, let f_u be a function with $\text{sup}(f_u \subseteq Z)$, and let f_a be a set of assignments to $Y \subseteq X$ (i.e., f_a is a function with $\text{sup}(f_a \subseteq Y)$). Then for an assignment t to Y , its extension t' to all variables X can improve on f_u (that is, $(\bigoplus_{j=1}^n f_j)(t) \downarrow_Z < f_u(t)$), if $\text{sink}(f_a \downarrow_Z, f_u \downarrow_Z)(t) \neq \top$.

Hence, the sinking operation generalizes the comparison of a lower bound to an upper bound by “filtering out” assignments that cannot improve on a bounding function f_u .

Algorithm 1 shows the pseudo-code for the resulting algorithm SBBTD (set-based branch-and-bound with tree decomposition). SBBTD is given a constraint f_a (corresponding to a set of current assignments and their values), a cluster v_i with a set of variables Y_{v_i} that remain to be assigned, and an upper bound function f_u whose support is a subset of the variables $\text{sep}(v_i)$ (f_u is a constant in the case where $v_i = v_{\text{root}}$). SBBTD returns a constraint corresponding to the extension of assignments f_a to solutions of the subproblem rooted at v_i (again, the result is a constant in the case where $v_i = v_{\text{root}}$). The valuation of this constraint is the value of best solution of the subproblem rooted at v_i , or a value greater than or equal to $f_u(t)$, if this best value is greater than or equal to $f_u(t)$. SBBTD uses two functions G_{v_i}, R_{v_i} to record the goods (solutions to the subproblem rooted at v_i) for each v_i . G_{v_i} is a soft constraint that contains the actual goods, while R_{v_i} is a hard constraint that contains the information whether an assignment has been recorded as a good or not (the use of two functions is necessary because a good can have any value in E , thus function G_{v_i} alone cannot give sufficient information whether the good has been stored or not). That is, an assignment t is recorded as a good for the separator if $R_{v_i}(t) = \top$, and not recorded if $R_{v_i}(t) = \perp$; in case the good is recorded, its value is $G_{v_i}(t)$. Initially, $R_{v_i} = G_{v_i} = \perp$.

SBBTD starts by filtering out the assignments whose value exceeds the bounding function (line 1). Inside a cluster (lines 19-28), SBBTD operates like branch-and-bound, except that it restricts variables to subsets of their domains (partition elements) instead of single values. Once all variables in the cluster have been assigned, SBBTD turns to its children (lines 3-17). SBBTD chooses a child v_j and first computes the subset of assignments f'_a of f_a that are not previously recorded as goods of $\text{sep}(v_j)$ (line 7). If there are any assignments

```

SBBTD( $f_a, v_i, Y_{v_i}, f_u$ )
1:  $f_a \leftarrow \text{sink}(f_a, f_u)$ 
2: if  $Y_{v_i} = \emptyset$  then
3:    $F \leftarrow \text{children}(v_i)$ 
4:   while  $F \neq \emptyset$  and  $f_a \neq \top$  do
5:     choose  $v_j \in F$ 
6:      $F \leftarrow F \setminus v_j$ 
7:      $f'_a \leftarrow R_{v_j} \oplus f_a$ 
8:     if  $f'_a \neq \top$  then
9:        $h_a \leftarrow \text{lift}(f'_a) \downarrow_{\text{sep}(v_j)}$ 
10:       $e_a \leftarrow \text{SBBTD}(h_a, v_j, \chi(v_j) \setminus \text{sep}(v_j), f_u \downarrow_{\text{sep}(v_j)})$ 
11:       $G_{v_j} \leftarrow G_{v_j} \oplus (e_a \oplus h_a)$ 
12:       $R_{v_j} \leftarrow R_{v_j} \oplus \text{compl}(h_a)$ 
13:    end if
14:     $f_a \leftarrow f_a \oplus G_{v_j}$ 
15:     $f_a \leftarrow \text{sink}(f_a, f_u)$ 
16:  end while
17:  return  $f_a \downarrow_{\text{sep}(v_i)}$ 
18: else
19:  choose  $x_i \in Y_{v_i}$ 
20:   $S \leftarrow P_i$ 
21:   $I \leftarrow \{f \in \lambda(v_i) : x_i \in \text{sup}(f), \text{sup}(f) \subseteq \text{sup}(f_a) \cup x_i\}$ 
22:  while  $S \neq \emptyset$  and  $f_a \neq \top$  do
23:    choose  $p \in S$ 
24:     $S \leftarrow S \setminus p$ 
25:     $f'_a \leftarrow f_a \oplus (x_i \in p) \oplus_{f \in I} f$ 
26:     $f_u \leftarrow \min(f_u, \text{SBBTD}(f'_a, v_i, Y_{v_i} \setminus \{x_i\}, f_u))$ 
27:  end while
28:  return  $f_u$ 
29: end if

```

Algorithm 1: Set-based branch-and-bound with tree decompositions

not recorded as goods, SBBTD solves the subproblem rooted at v_j for these assignments (line 10), and records the solutions as new goods (lines 11 and 12). It updates the values of the current assignments f_a (lower bounds) (line 14), and compares it with the current upper bounds (line 15). It continues with the next child (if any) or returns the solutions to the subproblem. The initial call to the algorithm is $\text{SBBTD}(\perp, v_{\text{root}}, \chi(v_{\text{root}}), \top)$.

Since the formulation of the algorithm is independent of how the domain partitions are defined, Fig. 1 actually defines a spectrum of algorithms that is parameterized by the domain partitions P_i for each variable. The limiting cases of the spectrum are $|P_i| = |d_i|$ and $|P_i| = 1$, corresponding to finest and coarsest granularity of the domain partitions, respectively. In the first case, the set of assignments f_a actually consists of a single assignment with value smaller than $\top$, and thus Alg. 1 becomes identical to branch-and-bound on tree decompositions (BTD). In the second case, the restrictions $x_i \in p$ yield constraints identical to $\perp$, and thus the search tree degenerates to a list. Hence, in this case the algorithm is backtrack-free and becomes identical to dynamic programming on the tree (called cluster-tree elimination (CTE) in [Kask *et al.*, 2003]). For the cases in between, that is, $1 < |P_i| < |d_i|$, hybrids of search and dynamic programming are obtained.

Theorem 1 For a COP (X, D, F) with a tree decomposition (T, χ, λ) and any domain partitions $P_1, P_2, \dots, P_n$ for variables $x_1, x_2, \dots, x_n$, the algorithm SBBTD is sound and complete, that is, $\text{SBBTD}(\perp, v_{\text{root}}, \chi(v_{\text{root}}), \top) = \alpha^*$.

For instance, consider the full adder example with the domain partitions $P_u, P_v = \{\{0\}, \{1\}\}$, $P_w, P_y = \{\{0,1\}\}$, and $P_{a1}, \dots, P_{a1} = \{\{G\}, \{S1, S2\}, \{U\}\}$. That is, the search simultaneously explores the values $\{0,1\}$ for w and y and the values $\{S1, S2\}$ for the mode variables. SBBTD starts by assigning the variables $\{u, v, w, y, a_1, a_2\}$ in the cluster v_1 , which gives two assignments, $\langle u, v, w, y, a_1, a_2 \rangle = \langle 0, 0, 0, 0, G, G \rangle$ and $\langle 0, 0, 1, 1, G, G \rangle$, both with value .95. SBBTD next considers a child of v_1 , for instance, v_2 . Since there are no goods recorded for this cluster so far, the solutions are computed for this subproblem for the assignments $\langle u, y \rangle = \langle 0, 0 \rangle$ and $\langle 0, 1 \rangle$. The value of these solutions are .0097 (corresponding to a S2 failure of Xor-gate 1) and .95, respectively. These solutions are recorded and combined with the two assignments, which now have the values .0092 and .90, respectively. Next, the solutions for the subproblem v_3 are computed simultaneously for the assignments $\langle v, w \rangle = \langle 0, 1 \rangle$ and $\langle 0, 1 \rangle$. The values are .95 and .01 (corresponding to a S1 failure of the Or-gate), respectively. After combining these solutions with the two assignments in v_1 , their value becomes .0088 and .018, respectively. Since there are no children left, SBBTD updates the bound for the best solution found to .018. This bound prunes all subsequent assignments to variables in v_1 except $\langle u, v, w, y, a_1, a_2 \rangle = \langle 1, 1, 0, 0, G, G \rangle$ and $\langle 1, 1, 1, 1, G, G \rangle$. Since the assignments $\langle u, y \rangle = \langle 1, 0 \rangle$ and $\langle 1, 1 \rangle$ for the subproblem v_2 lead to values worse than the bound, .018 is returned as optimal solution.

In comparison, observe that BTD (obtained as a specialization of SBBTD for the case $|P_i| = |d_i|$) suffers from the problem that it would explicitly iterate through all possible combinations of values for w and y until encountering the best assignment (which is obtained for $\langle w, y \rangle = \langle 1, 1 \rangle$); in contrast, SBBTD handles those combinations implicitly. Dynamic programming (obtained as a specialization of SBBTD for $|P_i| = 1$) suffers from the problem that it would consider more assignments than necessary (for example, it would compute the value for assignments involving $\langle a_1, a_2 \rangle = \langle U, U \rangle$, which is very low because it corresponds to a double fault).

4.3 Trade-off between Search and Symbolic Inference

SBBTD unifies search with good recording (BTD) and dynamic programming (CTE). In fact, the goods that the BTD algorithm computes can be understood as partial construction of the messages sent between clusters by the dynamic programming algorithm CTE [Kask *et al.*, 2003]. Thus, the two limiting cases can be understood as lazy and eager forms of dynamic programming, respectively: BTD computes solutions to subproblems only as far as required to compute the optimal solution, whereas CTE computes them completely.

It has been previously shown [Jégou and Terrioux, 2003] that BTD (i.e., lazy dynamic programming) outperforms CTE (i.e., eager dynamic programming). This is because search on single assignments exploits the upper bounds as rigorously as

possible, and therefore the least number of assignments will be explored. However, this argument is based on counting assignments and holds only if assignments are represented explicitly. The picture changes when using techniques such as ADDs that can manipulate sets of assignments implicitly. Hence, a tension is created between making the partition elements in SBBTD smaller or larger: in the former case, the advantage is that as few assignments are explored as possible, but the disadvantage is that less possibilities exist for exploiting commonalities between assignments. In the latter case, the disadvantage is that more assignments might be explored, but the advantage is that assignments can be abstracted into a more compact description and handled implicitly. Thus in many practical cases, the optimal granularity will lie in an intermediate point ($1 \leq |P_i| < |d_i|$) along the spectrum; SBBTD allows us to adapt to this trade-off.

5 Implementation and Experiments

We implemented SBBTD using the CUDD (Colorado University Decision Diagram) package [Somenzi, 2004], which provides a library of routines for manipulating ADDs.

We evaluated the performance of this prototype on random, binary Max-CSP problems. Max-CSP is a constraint optimization problem where the tuples of a constraint have cost 0 if the tuple is allowed, and cost 1 if the tuple is not allowed; the optimal solution corresponds to an assignment that violates a minimum number of constraints. For each instance, the support of the constraints was determined in order to derive the hypergraph (a graph in this case). Then, a tree decomposition of the hypergraph was computed using the min-fill heuristics. We choose random domain partitions, whose granularity we varied by setting a certain percentage P of (random) variables to the partition $|P_i| = 1$, and the other variables to the partition $|P_i| = |d_i|$. We ran the experiments on a Pentium 4 Windows PC with 1 GB of RAM; due space restrictions we give only summarized results here.

Consistent with results in [Jégou and Terrioux, 2003], we found the number of assignments explored for $P=0\%$ (BTD) to be in most cases several times smaller than the number of assignments explored for $P=100\%$ (dynamic programming). E.g., in an example class with $N=40$ variables, $C=80$ constraints, domain size $K=4$ and tightness $T=9$, the mean number of goods recorded for $P=0\%$ was 50,00 to 100,000, whereas for $P=100\%$ it was 250,000 to 1,000,000. However, the ADD representation of constraints typically achieved a compaction of around one to two orders of magnitude (consistent with observations in [Hoey *et al.*, 1999]). Therefore, and also because larger partition elements reduce the number of recursive calls, SBBTD ran faster for larger values of P in almost all cases. E.g., for $N=40$, $K=4$, $C=80$ and $T=9$, the mean runtime for $P=0\%$ was around 100 sec, but around 0.5 to 1 sec for $P=100\%$. Given that for $P=100\%$ the computation required up to one order of magnitude more memory than $P=0\%$, selecting an intermediate granularity ($0\% < P < 100\%$) allowed significant runtime improvements over BTD within still acceptable memory bounds. We are currently working on structured examples (like in [Jégou and Terrioux, 2003]) and real-world examples.

6 Conclusion

We presented an algorithm for solving soft constraint problems by generalizing branch-and-bound to search on sets of assignments and perform inference (dynamic programming) within those sets of assignments. This hybrid approach can exploit regularities in the constraints, while it can avoid memory explosion by controlling the size of the sets. In contrast to work in [Hoey *et al.*, 1999; Jensen *et al.*, 2002], our approach is more general in that it addresses valued constraint satisfaction problems [Schiex *et al.*, 1995], and incorporates a decomposition of the problem into several subproblems. Future work includes ways to automatically determine domain partitions (appropriate points in the spectrum), and augmenting the algorithm with symbolic versions of constraint propagation techniques [Cooper and Schiex, 2004] in order to further improve the bounds.

References

- [Bahar *et al.*, 1993] Iris Bahar, Erica Frohm, Charles Gaona, Gary Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *Proc. ICCAD-93*, pages 188–191, 1993.
- [Bryant, 1986] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
- [Cooper and Schiex, 2004] Martin Cooper and Thomas Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154:199–227, 2004.
- [Gottlob *et al.*, 2000] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.
- [Hoey *et al.*, 1999] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proc. UAI-99*, pages 279–288, 1999.
- [Jégou and Terrioux, 2003] Philippe Jégou and Cyril Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.
- [Jensen *et al.*, 2002] Rune Jensen, Randy Bryant, and Manuela Veloso. SetA*: An efficient BDD-based heuristic search algorithm. In *Proceedings AAAI-02*, 2002.
- [Kask *et al.*, 2003] Kalev Kask, Rina Dechter, and Javier Larrosa. Unifying cluster-tree decompositions for automated reasoning. Technical report, University of California at Irvine, 2003.
- [Schiex *et al.*, 1995] Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings IJCAI-95*, 1995.
- [Somenzi, 2004] Fabio Somenzi. CUDD release 2.4.0, 2004. <http://vlsi.colorado.edu/~fabio>.
- [Terrioux and Jégou, 2003] Cyril Terrioux and Philippe Jégou. Bounded backtracking for the valued constraint satisfaction problems. In *Proceedings CP-03*, 2003.

Solving Checkers*

J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller,
R. Lake, P. Lu and S. Sutphen

Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
jonathan@cs.ualberta.ca

Abstract

AI has had notable success in building high-performance game-playing programs to compete against the best human players. However, the availability of fast and plentiful machines with large memories and disks creates the possibility of *solving* a game. This has been done before for simple or relatively small games. In this paper, we present new ideas and algorithms for solving the game of checkers. Checkers is a popular game of skill with a search space of 10^{20} possible positions. This paper reports on our first result. One of the most challenging checkers openings has been solved – the White Doctor opening is a draw. Solving roughly 50 more openings will result in the game-theoretic value of checkers being determined.

1 Introduction

High-performance game-playing programs have been a major success story for AI. In games such as chess, checkers, Othello, and Scrabble, mankind has been humbled by the machine. However, although these programs are very strong, they can still lose a game to a human. They are strong, but not perfect. Perfection requires one to “solve” a game. Allis defines three levels of solving [Allis, 1994]:

1. Ultra-weakly solved. The game-theoretic value for the game has been determined. An ultra-weak solution is mainly of theoretical interest. For example, Hex is a first player win, but no one knows the winning strategy.
2. Weakly solved. The game is ultra-weakly solved and a strategy is known for achieving the game-theoretic value from the opening position, assuming reasonable computing resources. Several well-known games have been weakly solved, including Connect Four [Allis, 1988], Qubic [Allis, 1994], Go Moku [Allis, 1994], and Nine Men’s Morris [Gasser, 1996].
3. Strongly solved. For all possible positions, a strategy is known for determining the game-theoretic value for both players, assuming reasonable computing resources.

*The support of NSERC, iCORE and the University of Alberta is acknowledged.

Strongly solved games include 6-piece chess endgames and Awari [Romein and Bal, 2003].

Note the qualification on resources in the above definitions. In some sense, games such as chess are solved since the minimax algorithm can in principle determine the game-theoretic value, given a large enough amount of time. Resource constraints preclude such impractical “solutions”.

How difficult is it to solve a game? There are two dimensions to the difficulty [Allis *et al.*, 1991]: *decision complexity*, the difficulty required to make correct decisions, and *space complexity*, the size of the search space. Checkers is considered to have high decision complexity and moderate space complexity. All the games solved thus far have either low decision complexity (Qubic; Go-Moku), low space complexity (Nine Men’s Morris, size 10^{11} ; Awari, size 10^{12}) or both (Connect-Four, size 10^{14}).

Checkers, or 8×8 draughts, is popular in the British Commonwealth (present and past) and in the United States. The rules are simple (pieces move one square at a time diagonally, and can capture by jumping) and the number of piece types is small (kings and checkers), yet the game is quite challenging. Checkers has high decision complexity (more complex than 9×9 Go and the play of bridge hands, on par with backgammon, but less complex than chess) and moderate space complexity (10^{20} positions versus unsolved games such as backgammon, 10^{19} positions, and chess, 10^{44} positions). The best checkers programs are stronger than the best human players (e.g., CHINOOK won the World Man-Machine Championship [Schaeffer, 1997]).

The number of possible placings of checkers pieces on the board is roughly 5×10^{20} [Chinook, 1995]. This number is misleading since it includes positions that are not legally reachable from the start of the game. For example, although there are 9×10^{19} possible positions with 24 pieces on the board, only a small fraction can be reached in a game (e.g., 12 kings versus 12 kings is not possible).

Our effort to solve the game of checkers began in 1989! After almost 16 years of research and development, and countless thousands of computer hours, we are pleased to report the first major milestone in our quest. The White Doctor opening (shown in Figure 1) has been proven to be a draw.¹ This is a

¹Tournament checkers is played using the so-called “three-move ballot”. In competitions, games begin after the first 3 moves (3 ply)

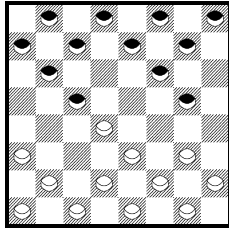


Figure 1: The White Doctor: White to play and only draw

difficult opening, as Black begins the game with a huge positional disadvantage—so large that humans consider Black’s best play to be to sacrifice a checker early on to relieve the pressure. Notably, this opening was played in the decisive game of the 1992 Tinsley–CHINOOK World Championship match [Schaeffer, 1997].

This paper has the following contributions:

1. A new hybrid combination of best-first and depth-first search results to the construction of minimax proof trees. Heuristic value iteration is introduced as a means of concentrating search effort where it is most needed.
2. A solution to the White Doctor opening. Achieving this result took years of computing, involving the pre-computation of 10^{13} values (endgame databases), and building the proof tree which took a further roughly 10^{13} positions. This is an important first step towards weakly solving the game of checkers.
3. The largest game-tree proof to date. This was made possible not only by our new hybrid search approach, but also by the integration of several state-of-the-art algorithms and enhancements into one coherent system.

2 Algorithm Overview

A formal proof of a game’s value can be done by depth-first alpha-beta ($\alpha\beta$) or best-first proof number search (PNS) [Allis, 1994]. For the large and complex checkers search space, neither turned out to be effective. Instead we propose a hybrid approach of the two: heuristically guided proofs.

The proof procedure has four algorithm/data components:

1. Endgame databases (backward search). Computations from the end of the game backward have resulted in a database of 4×10^{13} positions (≤ 10 pieces on the board) for which the game-theoretic value has been computed (strongly solved).
2. Proof tree manager (search management). This component maintains a tree of the proof in progress, traverses it, and generates positions that need to be explored to further the proof’s progress.
3. Proof solver (forward search). Given a position to search, this component uses two programs, $\alpha\beta$ and PNS, to determine the value of the position.

are randomly selected. Each starting position is played twice, with sides switched for the second game.

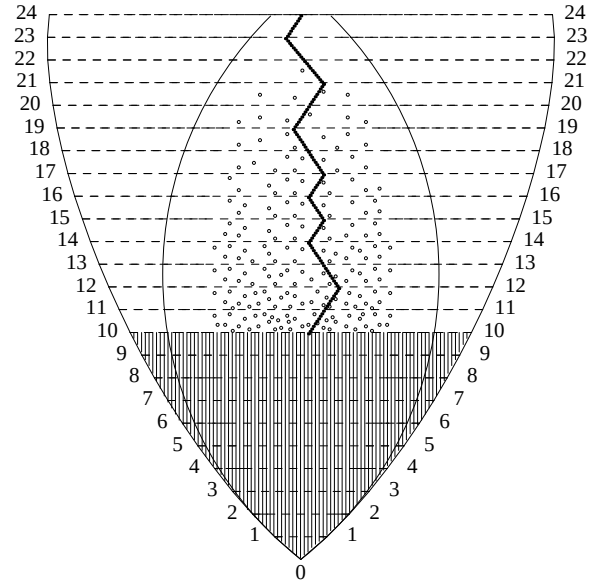


Figure 2: The checkers search space (log scale horizontally)

4. Seeding (expert input). From the human literature, a single line of “best play” is identified. This is fed to the proof tree manager as the initial line to explore.

Figure 2 illustrates this approach. It plots the number of pieces on the board (vertically) versus the logarithm of the number of positions (using data from [Chinook, 1995]; the widest point is with 23 pieces, 1.3×10^{20} positions). The endgame database phase of the proof is the shaded area, all positions with 10 or fewer pieces.

Seeding the proof process with an initial line of play is critical to the prover’s performance; it is shown in Figure 2 as a sequence of solid bold lines. The line leads from the start of the opening into the endgame databases. Seeding is not necessary to the proof—the prover is capable of doing the proof with no expert guidance. However, this single line of play allows the proof process to immediately start doing work that is likely to be relevant to the proof. Without it, we have seen the prover spend considerable effort flailing about while it tries to find the key lines of play.

The inner oval area in Figure 2 illustrates that only a portion of the search space is relevant to the proof. Positions may be irrelevant because they are unreachable in this opening, or are not required for the proof. The small circles illustrate positions with more than 10 pieces for which a value has been proven. Each of the circles represents the root of a search tree which has been solved. For clarity, the solved subtree below that root is not shown in the diagram.

3 Endgame Databases

Endgame databases were pioneered in chess [Thompson, 1986], and were instrumental in the success of the CHINOOK program [Schaeffer, 1997]. Using retrograde analysis, subsets of a game with a small number of pieces on the board can be exhaustively enumerated to compute which positions are

wins, losses or draws. Whenever a search reaches a database position, instead of using an inexact heuristic evaluation function, the exact value of the position can be retrieved.

We have computed all endgame databases up to 10 pieces on the board, for any combination of kings and checkers [van den Herik *et al.*, 2003]. The total database size is 39 trillion positions, roughly 4×10^{13} . Although this sounds impressive, it represents a paltry one ten-millionth of the total search space [Chinook, 1995]!

How valuable are the databases? Unfortunately, because of the win/loss/draw construction, a good measure of the difficulty of 10-piece endgames is not available—although it is presumed to be very high. Anecdotal evidence includes:

1. Perfect play databases for the 7-piece endgames have been computed, with the longest winning line being 253 ply [Trice and Dodgen, 2003].
2. [Schaeffer *et al.*, 2003] report that some of their database computations had winning lines in excess of 180 ply—just to force a checker advance or a piece capture.
3. In the 5-king versus 3-king and 2-checker endgame, the longest winning line is 279 ply [Gilbert, 2005].
4. Draws are much harder to prove than wins/losses.

This evidence strongly suggests that the databases are removing hundreds of ply from the search depth of the proof tree.

4 Search Algorithm

The proof search is split into a front-end proof tree manager and a back-end prover. The front-end manager maintains a large persistent proof tree, and selects, stores, and organizes the search results. Proofs of individual positions are done in parallel by a farm of back-end provers.

4.1 Proof Tree Manager (front end)

The proof tree manager starts off with an empty proof tree. The proof is “seeded” by being fed an initial line of play. This line is considered by humans to be best play by both sides. The proof manager starts at the last position in the line and solves it, then backs up one ply, solves that sub-tree, and so on until the root of the line is reached.

The proof is done iteratively, as shown in Figure 3. In minimax search, we have seen iterative algorithms based on search depth (iterative deepening) and confidence in the root value (as in conspiracy numbers [McAllester, 1988]). Our algorithm is novel in that it iterates on the range of relevant heuristic values for the proof.

Classical iterative deepening in game-playing programs is used for several reasons, one of which is to improve search efficiency by using the $\alpha\beta$ tree from iteration i as the basis for starting iteration $i+1$. The assumption is that increasing the depth does not cause substantial changes to the tree structure; each iteration refines the previous iteration’s solution. In contrast, without iterative deepening, the search can go off in the wrong direction and waste lots of effort before stumbling on the best line of play.

Our iterative value solution works for similar reasons. A position may require considerable effort to formally determine its proven value. However, the higher/lower the score

of the position, the more likely it is a win/loss. Rather than invest the effort to prove such a position, this work is postponed until it has been shown to be needed. We do this by introducing two new values to the proof tree. To win, loss, draw and unknown, we add *likely win* and *likely loss*.

All searches are done relative to a heuristic threshold. The back-end prover returns an assessment of a position. Any position exceeding the threshold is considered a likely win; any falling below the negative threshold is a likely loss. The proof manager repeatedly traverses the proof tree, identifies nodes to be searched, sends them off for assessment by the prover, and integrates the results. Any node with a score outside the threshold has a likely result and is not expanded further within this threshold; it is effectively treated as if it is proven. The thresholds can be loosely thought of as imposing a symmetric $\alpha\beta$ search window on the best-first proof manager. A value outside the window in $\alpha\beta$ means the result is irrelevant; a value outside the heuristic threshold means that we postpone the proof until we know the result will be relevant.

Once the proof tree is complete for a given heuristic threshold, the threshold is increased and the search restarted to construct a proof for the new threshold. Any position that was previously resolved to a real proven value remains proven, and positions that have a heuristic score that remain outside the new heuristic threshold remain likely wins/losses. Only non-proven positions with heuristic values inside the new threshold are now considered, and they must be re-searched with the new heuristic limit. This iterative process continues until the heuristic threshold exceeds the largest possible heuristic value. At this point all likely wins/losses have been resolved, and the proof tree is complete.

Ideally, after the first likely proof tree is constructed, the work would consist solely of trying to prove that the likely wins/losses are even more likely wins/losses. In practice, the heuristics we use are occasionally wrong (they are, after all, only heuristics), and a proof tree for a position may grow large enough that other previously less desirable positions become a more attractive alternative for expansion.

For the White Doctor, we used settings of MIN = 125 (a checker is worth 100 points) and INC = 5 in Figure 3. These might not be the best choices; we are experimenting with different values on other checkers openings.

The proof manager uses PNS, trying to answer two questions about the value of the proof tree: “is it at least a likely win”, and “is it at most a likely loss.” A likely win is assigned a proof number of 0 for the is-likely-win question, a likely loss is assigned a proof number of 0 for the is-likely-loss question, and a draw is assigned a disproof number of 0 for both questions. The prover tries to investigate the two questions simultaneously by selecting nodes for consideration that (ideally) contribute to answering both.

The proof tree manager traverses the proof tree and identifies nodes that are needed or likely to be needed in the proof. While a traditional PNS algorithm expands one node at a time, our manager expands many promising nodes (possibly hundreds of them), to have enough work to keep multiple back-end prover processors busy.

When a node is selected for expansion, it is added to a work-to-do file that is used as input to the prover pro-

```

ProofManager( Seeding[] ) {
  // Seeding: seeding line for the proof
  // Seeding[ 1 ] is the opening position
  // MIN: starting proof heuristic threshold
  // WIN: ending proof threshold
  // INC: heuristic increment between likely proofs

  // Iterate on heuristic threshold
  for ( thresh = MIN; thresh <= WIN; thresh += INC )
    // Iterate over the seeded line
    for ( move = Length( Seeding ); move > 0; move -= 1 ) {
      pos = Seeding[ move ];

      // Solve current position to threshold
      repeat {
        worklist = PNSearch( pos, thresh );
        if( NotEmpty( worklist ) ) {
          StartProvers( worklist, results, thresh );
          WaitForResults( worklist, results );
          UpdateProofTree( pos, results, thresh );
        }
      } until ( IsEmpty( worklist ) );
    }
  }
}

```

Figure 3: Proof manager

cesses. The back-end is given information for the generation of heuristic proof numbers and any proven information that is known about the position (e.g., it might already be known that it is not a loss).

The proof manager saves all back-end search results so that they can be re-used for searches with a different heuristic threshold (or from a different opening). A persistent B-tree is used to store information on every position examined in every opening, including both the back-end results and the backed-up values computed by the front end.

In the cases where the back-end proof search does not provide enough information to answer the front end’s question of interest, the heuristic score from the back end is used to generate heuristic proof and disproof numbers. CHINOOK is used to provide the heuristic scores. If the $\alpha\beta$ search result is small enough to initiate the proof search, then the result is used to help initialize the proof numbers. The proof number for a win is a function of: 1) the difference between the heuristic score and the heuristic threshold (a high score is more likely to be a win than an even score), and 2) a penalty is added for positions where both sides have a king, as these tend to be much harder to prove.

The disproof number for a loss is the same as the proof number for a win, with one exception. We use one additional statistic from CHINOOK. If the principal variation of the $\alpha\beta$ search result depends on a draw from a database lookup, we decrease the disproof number. Proof numbers for a loss and disproof numbers for a win are defined similarly. As a final modification, the numbers that depend on the minimum of their child values (as opposed to the sum) are divided by the average branching factor.

4.2 Proof Solver (back end)

Pseudo-code for the back-end prover is shown in Figure 4. It is a novel combination of a traditional heuristic $\alpha\beta$ search and a proof search. The (cheap) heuristic search value is used to determine whether the current position is relevant to the cur-

```

Prover() {
  repeat {
    GetWork( pos, thresh );

    // HeuristicSearch returns alpha-beta heuristic value
    heur = HeuristicSearch( pos, thresh );

    // If value inside threshold range, do PNSearch
    lower = LOSS; upper = WIN;
    if( -thresh*2 < heur && heur < 2*thresh ) {
      // This is a simplified version of the actual code
      // Check for WIN, at most DRAW, or UNKNOWN
      result = DFPNSearch( pos, IS_WIN );
      if( result == PROVEN ) { lower = WIN ; }
      else if( result == DISPROVEN ) { upper = DRAW; }

      if( lower != upper && time remaining ) {
        // Check for LOSS, at least DRAW, or UNKNOWN
        result = DFPNSearch( pos, IS_LOSS );
        if( result == PROVEN ) { upper = LOSS; }
        else if( result == DISPROVEN ) { lower = DRAW; }
      }
    }
    SaveResult( pos, thresh, heur, lower, upper );
  }
}

```

Figure 4: Proof solver process

rent proof threshold. The (expensive) proof search attempts to formally prove the result of the current position. The latter is only done if the former indicates it is needed.

Each back-end position is first searched by an $\alpha\beta$ searcher using a heuristic evaluation function and the endgame databases. The search is limited to 25 seconds, reaching a depth of 17 to 23 ply plus search extensions. The heuristic value returned is compared to the search threshold. Values outside the threshold range are considered as likely wins/losses by the proof tree manager. If the heuristic value is more than twice the size of the threshold, then no further processing is done; the formal proof search is postponed until later in the proof when we will have more confidence that we really need this result.

The prover uses Nagai’s Df-pn⁺ algorithm [Nagai, 2002], a depth-first search variant of best-first PNS. Df-pn⁺ was preferred over PNS, in part, because of memory concerns. The algorithm’s space requirements are limited by the size of the transposition table. In contrast, PNS needs to store the entire tree in memory. This gives Df-pn⁺ an important advantage, since memory is also needed to reduce the frequency of costly disk I/O caused by endgame database accesses.

The time limit for the Df-pn⁺ component is 100 seconds per search. Most of the search time is spent trying to answer the “easy” question, the opposite result to that of the heuristic search. For example, if the heuristic result indicates that the side to move has a large advantage (a possible win), the prover first tries to prove or disprove the position as a loss. This was done because this was usually the easier question to answer and a disproof of a loss or win may be a sufficient answer for the proof manager given that we expect the value of the proof to be a draw. If a disproof for the question is found in less than the maximum time allotted, another proof search is started to answer the other proof question.

Proving the value of a node implies finding a path to the endgame databases (or a draw by repetition). Clearly, the fewer pieces on the board, the “closer” one is to a database

position and the easier it should be to (dis)prove. In addition, it should be much easier to prove a win or disprove a loss if one side has more pieces than the other side. We combine these two factors to initialize the proof and disproof numbers, giving more attention to fewer pieces and favorable positions. Furthermore, because the standard Df-pn⁺ algorithm has a fundamental problem of computing (dis)proof numbers when position repetitions are present, the techniques in [Kishimoto and Müller, 2003] are incorporated into the solver.

4.3 Graph History Interaction

A forward-search-based proof cannot be correct unless the Graph History Interaction problem (GHI) is properly addressed. GHI can occur when a search algorithm caches and re-uses a search result that depends on the move history. In checkers, repeated positions are scored as a draw. It is possible that the value of a search is influenced by a draw-by-repetition score, and this result is saved in the proof tree or in a transposition table. If this cached result is later reached by a different move sequence, it might not be correct—we do not know if the draw-by-repetition scores are correct.

In the back-end prover, GHI is correctly handled by utilizing the new technique described in [Kishimoto and Müller, 2004]. GHI is avoided in the boundary between the front-end tree and the back-end searches by not communicating any move history information to the back-end. Finally, in the front-end manager, GHI is avoided by not saving any value in the proof tree that is unsafely influenced by a draw-by-repetition score somewhere in its subtree. This may cause the manager to spend extra time searching the tree, but no extra prover searches need be performed, as the prover results stored in the tree are path independent.

4.4 Implementation Insights

The effort to turn likely wins and likely losses into proven values takes the vast majority of search effort. This holds even for a fairly high initial heuristic threshold, where likely wins/losses are positions that human checkers experts agree are “game over”. The major reason for this is postponing moves; a non-winning side always takes the path of maximum resistance. For example, the losing side in a position is able to make terrible moves, such as sacrificing a checker for no apparent gain, to postpone reaching the endgame databases; the unknown aspect of a position with more than 10 pieces on the board (no matter how hopeless the position) is preferable to a known database loss. These lopsided positions must still be chased down to reach a proven conclusion.

The proof process is computationally demanding. Some of the performance issues include:

1. The proof process spends most of its time accessing the endgame databases. The cost can be reduced by locality of disk accesses and caching. Smart organization of the endgame databases and use of a 4GB machine (most of the memory going to I/O caching), means that 99% of all endgame database positions do not require I/O.
2. Despite the above performance optimizations, the proof process is still I/O intensive—keeping the I/O subsystem at maximum capacity. It is rare to see the CPU usage

beyond 10%. Many machine architectures do not have the hardware reliability for this sustained I/O demand.

3. The massive I/O means that we can only use local I/O; network I/O is too slow. That limits the machines that we can add to the computation to those that have 300 GB of available local disk.
4. I/O errors. Our programs are instrumented to detect and recover (if possible) from most types of I/O errors.

5 Results

The White Doctor opening was chosen as the first candidate to solve because of its importance in tournament practice to the checkers-playing community. The opening is one of the most difficult of the three-move ballots, with Black starting off with a large disadvantage. Many decades of human analysis determined that Black’s desperate situation meant that a checker had to be sacrificed to relieve the pressure. In a typical game, Black is down material for most of the game, barely hanging on to salvage a draw. Is the human analysis correct?

The White Doctor proof began in November 2003 and a heuristic proof for a threshold of 125 was completed one month later. For the next eight months, computer cycles were spent increasing the heuristic threshold to a win and verifying that there were no bugs in the proof.² In August 2004, we thought the proof was complete, but an inspection showed that some trivial positions with an advantage of 7 or more checkers had been eliminated from the proof (an accidental, historical error). This was corrected and in January 2005, we completed the last of the unresolved positions.

Is the proof correct? The endgame databases are being verified by Ed Gilbert, the author of the KINGSROW checkers program. He has independently verified that our 9-piece databases and 30% of our 10-piece databases are correct (his computation is ongoing). The front-end White Doctor proof tree is consistent and correct, assuming that all the back-end proofs returned the right answer. Many of the back-end values have been re-verified by a different solver developed as part of this project, increasing our confidence that the proof is indeed correct. The proof is available online at <http://www.cs.ualberta.ca/~chinook>. Several strong players have looked at the proof lines and found no errors.

Some interesting statistics on the proof include:

1. Number of positions given to the back-end to solve (excluding work that was used for experiments or shown to have bugs): 841,810.
2. Number of positions in one minimal proof tree: 223,178. The minimal proof tree is not a minimum proof tree. We did not try to maximize the usage of transpositions or choose the branches leading to the smallest subtree in generating the proof tree.
3. Longest line searched in the front-end proof tree: 67 ply.
4. Longest line in our minimal proof tree: 46 ply. The leaf node in the tree is the result of a 100 second proof number search (to a variable depth, possibly as much as 45

²Several minor problems were found, necessitating re-doing some of the calculations.