SEARCH, SATISFIABILITY, AND CONSTRAINT SATISFACTION PROBLEMS

SEARCH, SATISFIABILITY, AND CONSTRAINT SATISFACTION PROBLEMS

SEARCH

Incomplete Tree Search using Adaptive Probing

Wheeler Ruml
Division of Engineering and Applied Sciences
Harvard University
33 Oxford Street, Cambridge, MA 02138 USA
ruml@eecs.harvard.edu

Abstract

When not enough time is available to fully explore a search tree, different algorithms will visit different leaves. Depth-first search and depth-bounded discrepancy search, for example, make opposite assumptions about the distribution of good leaves. Unfortunately, it is rarely clear *a priori* which algorithm will be most appropriate for a particular problem. Rather than fixing strong assumptions in advance, we propose an approach in which an algorithm attempts to adjust to the distribution of leaf costs in the tree while exploring it. By sacrificing completeness, such flexible algorithms can exploit information gathered during the search using only weak assumptions. As an example, we show how a simple depth-based additive cost model of the tree can be learned on-line. Empirical analysis using a generic tree search problem shows that adaptive probing is competitive with systematic algorithms on a variety of hard trees and outperforms them when the node-ordering heuristic makes many mistakes. Results on boolean satisfiability and two different representations of number partitioning confirm these observations. Adaptive probing combines the flexibility and robustness of local search with the ability to take advantage of constructive heuristics.

1 Introduction

Consider the problem of searching a finite-depth tree to find the best leaf. Many search trees arising in practical applications are too large to be explored completely. Given a limited amount of time, one can only hope to search the tree in such a way that leaves with a greater chance of being optimal are encountered sooner. For instance, when a node-ordering heuristic is available, a depth-first search can expand the children of a node in the order in which they are preferred by the heuristic. However, the backtracking order of depth-first search will visit the second-ranked child of the last internal branching node before reconsidering the choice at the next to last branching node. Each decision at which a non-preferred child is chosen is called a discrepancy [Harvey and Ginsberg, 1995]. Depth-first search will visit the leaf whose path from the root has all discrepancies below depth i before visiting the leaf with a single discrepancy at depth i. This corresponds to an implicit assumption that a single discrepancy at depth i will lead to a worse leaf than taking discrepancies at every deeper depth.

Limited discrepancy search [Harvey and Ginsberg, 1995; Korf, 1996] was designed with a different assumption in mind. It assumes that discrepancies at any depth are equally disadvantageous and so visits all leaves with k discrepancies anywhere in their paths before visiting any leaf with $k + 1$ discrepancies. Depth-bounded discrepancy search [Walsh, 1997] uses a still different assumption: a single discrepancy at depth i is worse than taking discrepancies at all depths shallower than i. Motivated by the idea that node-ordering heuristics are typically more accurate in the later stages of problem-solving, when local information better reflects the remaining subproblem, this assumption is directly opposed to the one embodied by depth-first search.

When faced with a new search problem, it is often not obvious which algorithm's assumptions most accurately reflect the distribution of leaf costs in the tree or even if any of them are particularly appropriate. In this paper, we investigate an adaptive approach to tree search in which we use the costs of the leaves we have seen to estimate the cost of a discrepancy at each level. Simultaneously, we use these estimates to guide search in the tree. Starting with no preconceptions about the relative advantage of choosing a preferred or non-preferred child node, we randomly probe from the root to a leaf. By sharpening our estimates based on the leaf costs we observe and choosing children with the probability that they lead to solutions with lower cost, we focus the probing on areas of the tree that seem to contain good leaves.

This stochastic approach is incomplete and cannot be used to prove the absence of a goal leaf. In addition, it generates the full path from the root to every leaf it visits, incurring overhead proportional to the depth of the tree when compared to depth-first search, which generates roughly one internal node per leaf. However, the problem-specific search order of adaptive probing has the potential to lead to better leaves much faster. Since an inappropriate search order can trap a systematic algorithm into exploring vast numbers of poor leaves, adaptive probing would be useful even if it only avoided such pathological performance on a significant fraction of problems.

After describing the details of an algorithm based on this adaptive approach, we investigate the algorithm's performance using the abstract tree model of Harvey and Ginsberg [1995]. We find that adaptive probing outperforms systematic methods on large trees when the node-ordering heuristic is moderately inaccurate, and exhibits better worst-case performance whenever the heuristic is not perfect at the bottom of the tree. To confirm these observations, we also test the algorithm on two different representations of the combinatorial optimization problem of number partitioning and on the goal-search problem of boolean satisfiability. It performs well on satisfiability and the naive formulation of number partitioning, but is competitive only for long run-times when using the powerful Karmarkar-Karp heuristic.

2 An Adaptive Probing Algorithm

Within the general approach outlined above, there are many ways to extract information from observed leaf costs. In this paper, we will evaluate one simple model. We will assume that the cost of every leaf is the sum of the costs of the actions taken to reach it from the root. Each position in the ordered list of children counts as a distinct action and actions at different levels of the tree are modeled separately. So a tree of depth d and branching factor b requires db parameters, one for each action at each level. The model assumes, for instance, that the effects of choosing the second-most-preferred child at level 23 is the same for all nodes at level 23. This is just a generalization of the assumption used by discrepancy search algorithms. In addition, we will estimate the variance of the action costs by assuming that each estimated cost is the mean of a normal distribution, with all actions having the same variance.

This model is easy to learn during the search. Each probe from the root corresponds to a sequence of actions and results in an observed leaf cost. If $a_j(i)$ is the cost of taking action i at depth j and l_k is the cost of the kth leaf seen, probing three times in a binary tree of depth three might give the following information:

$$\begin{array}{cccccccc} a_0(0) & + & a_1(0) & + & & a_2(1) & = l_0 \\ a_0(0) & + & & a_1(1) + & a_2(0) & & = l_1 \\ & a_0(1) + & a_1(0) & + & a_2(0) & & = l_2 \end{array}$$

We can then estimate the $a_j(i)$ using a least squares regression algorithm. In the experiments reported below, a perceptron was used to estimate the parameters [Cesa-Bianchi *et al.*, 1996]. This simple gradient descent method updates each cost according to the error between a prediction of the total leaf cost using the current action estimates, $\hat{l}_k$, and the actual leaf cost, l_k. If d actions were taken, we update each of their estimates by

$$\eta \frac{(l_k - \hat{l}_k)}{d}$$

where η controls the learning rate (or gradient step-size). All results reported below use $\eta = 0.2$, although similar values also worked well. (Values of 1 and 0.01 resulted in reduced performance.) This update requires little additional memory, takes only linear time, adjusts d parameters with every leaf, and often performed as well as an impractical $O(d^3)$ singular value decomposition estimator. It should also be able to track changes in costs as the probing becomes more focussed, if necessary.

Because we assume that it is equal for all actions, the variance is straightforward to estimate. If we assume that the costs of actions at one level are independent from those at another, then the variance we observe in the leaf costs must be the sum of the variances of the costs selected at each level. The only complication is that the variance contributed by each level is influenced by the mean costs of the actions at that level—if the costs are very different, then we will see variance even if each action has none. More formally, if X and Y are independent and normally distributed with common variance σ^2_{XY}, and if W takes its value according to X with probability p and Y with probability $1-p$, then

$$\begin{aligned} \sigma^2_W &= E(W^2) - \mu^2_W \\ &= p(\mu^2_X + \sigma^2_{XY}) + (1-p)(\mu^2_Y + \sigma^2_{XY}) - \\ &\quad (p\mu_X + (1-p)\mu_Y)^2 \\ &= \sigma^2_{XY} + p\mu^2_X + (1-p)\mu^2_Y - (p\mu_X + (1-p)\,\mu_Y)^2 \end{aligned}$$

Since we can easily compute p by recording the number of times each action at a particular level is taken, and since the action costs are estimates of the μ_i, we can use this formula to subtract away the effects of the different means. Following our assumption, we can then divide the remaining observed variance by d to distribute it equally among all levels.

Using the model during tree probing is also straightforward. If we are trying to minimize the leaf cost, then for each decision, we want to select the action with the lower expected cost (i.e., the lower mean). As our estimates may be quite inaccurate if they are based on few samples, we don't always want to select the node with the lower estimated cost. Rather, we merely wish to select each action with the probability that it is truly best. Given that we have estimates of the means and variance of the action costs and we know how many times we have tried each action, we can compute the probability that one mean is lower than another using a standard test for the difference of two sample means. We then choose each action according to the probability that its mean cost is lower. To eliminate any chance of the algorithm converging to a single path, the probability of choosing any action is clamped at $0.05^{1/d}$ for a depth d tree, which ensures at least one deviation on 95% of probes.

Now we have a complete adaptive tree probing algorithm. It assumes the search tree was drawn from a simple model of additive discrepancy costs and it learns the parameters of the tree efficiently on-line. Exploitation of this information is balanced with exploration according to the variance in the costs and the number of times each action has been tried. The method extends to trees with large and non-uniform branching factors and depths. The underlying model should be able to express assumptions similar to those built into algorithms as diverse as depth-first search and depth-bounded discrepancy search, as well as many other weightings not captured by current systematic methods.

3 Empirical Evaluation

We first investigate the performance of this adaptive probing algorithm using an abstract model of heuristic search. This gives us precise control over the density of good nodes and the accuracy of the heuristic. To ensure that our conclusions apply to more complex domains, we will also evaluate the algorithm using two NP-complete search problems: the combinatorial optimization problem of number partitioning and the goal-search problem of boolean satisfiability.

3.1 An Abstract Tree Model

In this model, introduced by Harvey and Ginsberg [1995] for the analysis of limited discrepancy search, one searches for goal nodes in a binary tree of uniform depth. Goals are distributed according to two parameters: m, which controls goal density, and p, which controls the accuracy of the heuristic. Each node either has a goal below it, in which case it is *good*, or does not, in which case it is *bad*. Clearly, the root is good and bad nodes only have bad children. The probabilities of the other configurations of parent and children are:

P(good $\rightarrow$ good good) = $1 - 2m$
P(good $\rightarrow$ bad good) = $1 - p$
P(good $\rightarrow$ good bad) = $2m - (1 - p)$

The expected number of goal nodes is $(2 - 2m)^d$, where d is the depth of the tree.

Following Walsh's [1997] analysis of depth-bounded discrepancy search, we will estimate the number of leaves that each algorithm must examine before finding a goal using empirical measurements over lazily (but deterministically) generated random trees. To provide a leaf cost measure for adaptive probing, we continue the analogy with constraint satisfaction problems that motivated the model and define the leaf cost to be the number of bad nodes in the path from the root. (If we were able to detect failures before reaching a leaf, this would be the depth remaining below the prune.) The results presented below are for trees of depth 100 in which $m = 0.1$. The probability that a random leaf is a goal is 0.000027. By investigating different values of p, we can shift the locations of these goals relative to the paths preferred by the heuristic.

Figure 1 shows the performance of depth-first search (DFS), Korf's [1996] improved version of limited discrepancy search (ILDS), depth-bounded discrepancy search (DDS), and adaptive probing on 2,000 trees. A heuristic-biased probing algorithm is also shown. This algorithm selects the preferred child with the largest probability that would be allowed during adaptive probing. Following Walsh, we raise the accuracy of the heuristic as depth increases. At the root, $p = 0.9$ which makes the heuristic random, while at the leaves $p = 0.95$ for 75% accuracy. ILDS was modified to incorporate this knowledge and take its discrepancies at the top of the tree first.

Adaptive probing quickly learns to search these trees, performing much better than the other algorithms. Even though DDS was designed for this kind of tree, its assumptions are too strong and it always branches at the very top of the tree. ILDS wastes time by branching equally often at the bottom where the heuristic is more accurate. The *ad hoc* biased probing algorithm, which branches at all levels, is competitive

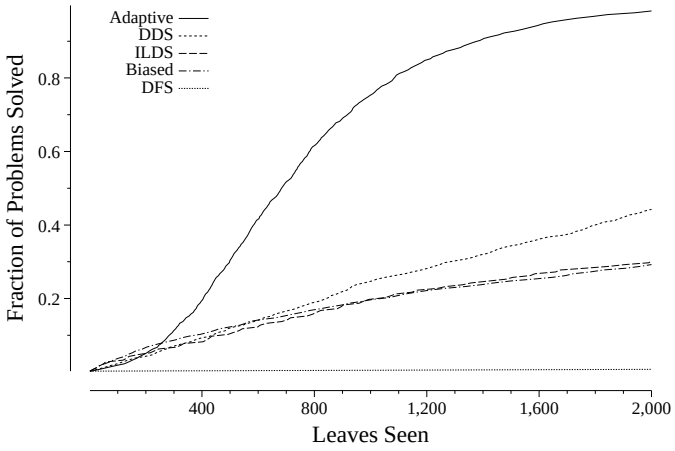

Figure 1: Probability of finding a goal in trees of depth 100 with m = 0.1 and p linearly varying between 0.9 at the root and 0.95 at the leaves.

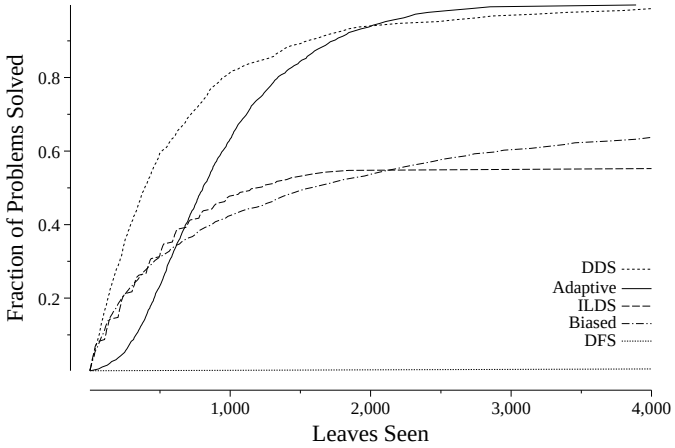

Figure 2: Performance on trees of depth 100, m = 0.1, and p varying from 0.9 at the root to 0.98 at the leaves.

with ILDS (and will actually surpass it, given more time) but fails to exploit the structure in the search space. DFS vainly branches at the bottom of the tree, ignorant of the fatal mistake higher in the tree, and solves almost no problems within 2,000 leaves.

DDS does better when the heuristic is more accurate, since its steadfast devotion to the preferred child in the middle and bottom of the tree is more often correct. Figure 2 shows the algorithms' performance on similar trees in which the heuristic is accurate 90% of the time at the leaves. DDS has better median performance, although adaptive probing exhibits more robust behavior, solving all 2,000 problems within 4,000 leaves. DDS had not solved 1.4% of these problems after 4,000 leaves and did not complete the last one until it had visited almost 15,000 leaves. In this sense, DDS has a heavier tail in its cost distribution than adaptive probing. Similar results were obtained in trees with uniform high p. Adaptive probing avoids entrapment in poor parts of the tree at the expense of an initial adjustment period.

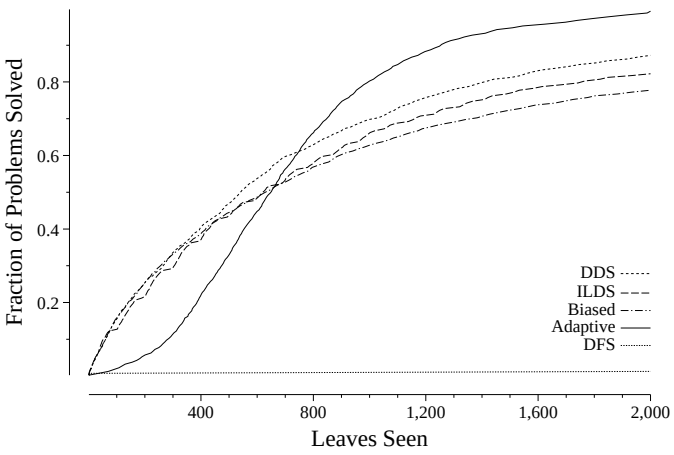

Figure 3: Performance on trees of depth 100, m = 0.1, and p varying from 0.98 at the root to 0.9 at the leaves.

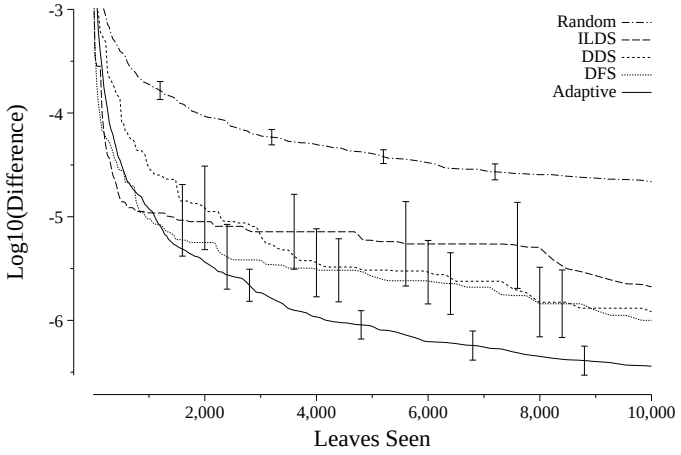

Figure 4: Searching the greedy representation of number partitioning. Error bars indicate 95% confidence intervals around the mean over 20 instances, each with 128 44-digit numbers.

Even with an accurate heuristic, however, the assumptions of DDS can be violated. Figure 3 shows what happens in trees in which the heuristic is accurate at the top of the tree and random at the very bottom. DDS still has an advantage over ILDS because a single bad choice can doom an entire subtree, but adaptive probing learns a more appropriate strategy.

To ensure that our insights from experiments with the abstract tree model carry over to other problems, we also evaluated the algorithms on three additional kinds of search trees.

3.2 Number Partitioning

The objective in a number partitioning problem is to divide a given set of numbers into two disjoint groups such that the difference between the sums of the two groups is as small as possible. It was used by Johnson et al. to evaluate simulated annealing [1991], Korf to evaluate his improvement to limited discrepancy search [1996], and Walsh to evaluate depth-bounded discrepancy search [1997]. To encourage difficult search trees by reducing the chance of encountering a perfectly even partitioning [Karmarkar *et al.*, 1986], we used instances with 64 25-digit numbers or 128 44-digit numbers.[1] (Common Lisp, which provides arbitrary precision integer arithmetic, was used to implement the algorithms.) Results were normalized as if the original numbers had been between 0 and 1. To better approximate a normal distribution, the logarithm of the partition difference was used as the leaf cost.

The Greedy Representation

We present results using two different representations of the problem. The first is a straightforward greedy encoding in which the numbers are sorted in descending order and then each decision places the largest remaining number in a partition, preferring the partition with the currently smaller sum. Figure 4 compares the performance of adaptive tree probing with depth-first search (DFS), improved limited discrepancy search (ILDS), depth-bounded discrepancy search (DDS),

[1]These sizes also fall near the hardness peak for number partitioning [Gent and Walsh, 1996], which specifies $log_{10}2^n$ digits for a problem with n numbers.

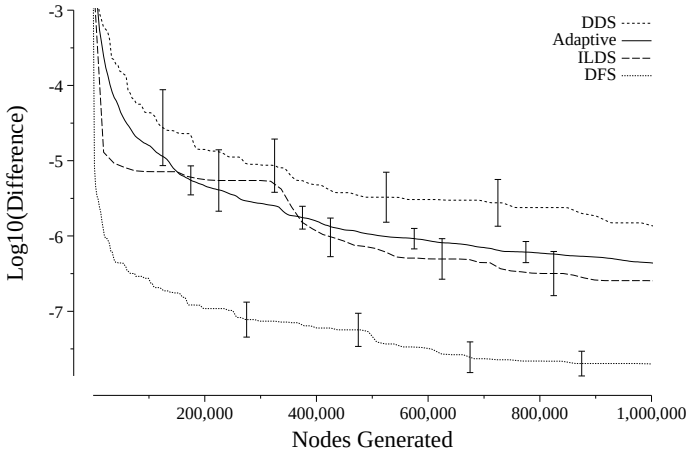

Figure 5: Performance on the greedy representation of number partitioning as a function of nodes generated.

and completely random tree probing. To provide a comparison of the algorithms' search orders, the horizontal axis represents the number of leaves seen. Adaptive probing starts off poorly, like random sampling, but surpasses all other algorithms after seeing about 1,000 leaves. It successfully learns an informative model of the tree and explores the leaves in a more productive order than the systematic algorithms.

However, recall that adaptive tree probing suffers the maximum possible overhead per leaf, as it generates each probe from the root. (This implementation did not attempt to reuse initial nodes from the previous probe.) The number of nodes (both internal and leaves) generated by each algorithm should correlate well with running time in problems in which the leaf cost is computed incrementally or in which the node-ordering heuristic is expensive. Figure 5 compares the algorithms on the basis of generated search nodes. (To clarify the plot, DFS and ILDS were permitted to visit many more leaves than the other algorithms.) In a demonstration of the importance of overhead, DFS dominates all the other algorithms in

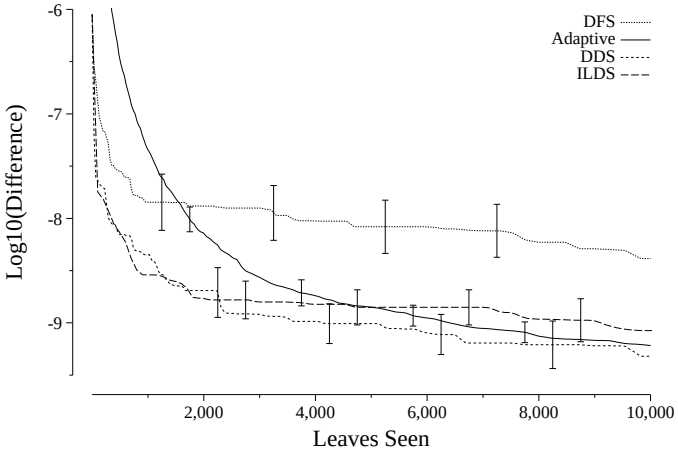

Figure 6: Searching the CKK representation of number partitioning. Each instance had 64 25-digit numbers.

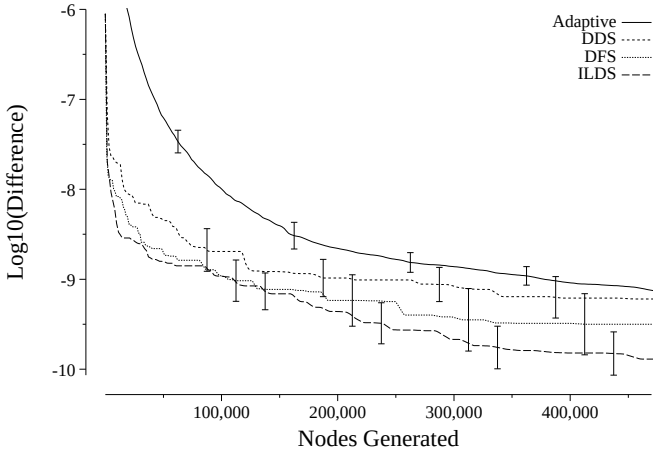

Figure 7: Performance on the CKK representation of number partitioning as a function of nodes generated.

this view, and ILDS performs comparably to adaptive probing. DFS reuses almost all of the internal nodes on each leaf's path, generating only those just above the leaves. Since ILDS needs to explore discrepancies at every level of the tree, it will usually need to generate a significant fraction of the path down to each leaf. DDS, which limits its discrepancies to the upper levels of the tree, incurs overhead similar to that of adaptive probing because it never reuses internal nodes in the middle of the tree.

On instances using 64 numbers, adaptive probing again dominated DDS, but was clearly surpassed by ILDS. (It performed on par with a version of ILDS that visited discrepancies at the top of the tree before those at the bottom.) This suggests that, in these search trees, the advantage of adaptive probing over ILDS and DDS increases with problem size.

The CKK Representation

A more sophisticated representation for number partitioning was suggested by Korf [1995], based on the heuristic of Karmarkar and Karp [1982]. The essential idea is to postpone the assignment of numbers to particular partitions and merely constrain pairs of number to lie in either different bins or the same bin. Numbers are considered in decreasing order and constrained sets are reinserted in the list according to the remaining difference they represent. This representation creates a very different search space from the greedy heuristic.

Figure 6 shows the performance of the algorithms as a function of leaves seen. DDS has a slight advantage over ILDS, although adaptive probing is eventually able to learn an equally effective search order. DFS and random sampling too often go against the powerful heuristic. As in the greedy representation, however, interior node overhead is an important consideration. Figure 7 shows that DDS and adaptive probing are not able to make up their overhead, and results using 128 numbers suggest that these difficulties increase on larger problems. Bedrax-Weiss [1999] argues that the KK heuristic is extraordinarily effective at capturing relevant information and that little structure remains in the space. These results are consistent with that conclusion, as the uniform and limited discrepancies of ILDS appear best.

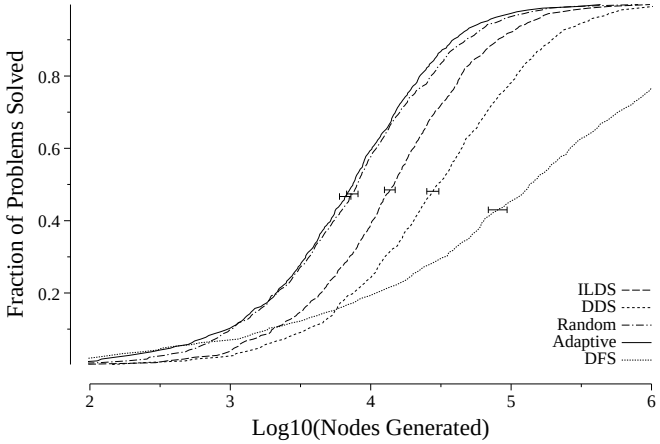

Figure 8: Fraction of random 3-satisfiability problems solved. Error bars indicate 95% confidence intervals around the mean over 1000 instances, each with 200 variables and 3.5 clauses per variable. (The DFS and DDS means are lower bounds.)

3.3 Boolean Satisfiability

We also tested on instances of boolean satisfiability. Following Walsh [1997], we generated problems according to the random 3-SAT model with 3.5 clauses per variable and filtered out any unsatisfiable problems. All algorithms used unit propagation, selected the variable occurring in the most clauses of minimum size, and preferred the value whose unit propagation left the most variables unassigned. The cost of a leaf was computed as the number of variables unassigned when the empty clause was encountered.

Figure 8 shows the percentage of 200-variable problems solved as a function of the number of nodes generated. Although Walsh used these problems to argue for the suitability of DDS, we see that both ILDS and purely random sampling perform significantly better. (Crawford and Baker [1994] similarly found random sampling effective on scheduling problems that had been converted to satisfiability

problems.) DFS performs very poorly. Adaptive probing performs slightly better than random sampling (this is most noticeable at the extremes of the distribution). Although slight, this advantage persisted at all problem sizes we examined (100, 150, 200, and 250 variables).

To summarize: in each search space we examined, the systematic search algorithms ranked differently in performance. This makes it difficult to select which algorithm to use for a new problem. Even when taking its overhead into account, adaptive probing seemed to perform respectably in every search space. The only space in which it was not the best or near the best was the CKK space for number partitioning, in which the node-ordering heuristic is very accurate. Of course, further work is needed to assess its performance in very different domains, such as those with a high branching factor, and against additional methods, such as interleaved depth-first search [Meseguer, 1997].

4 Related Work

Abramson [1991] used random sampling in two-player game trees to estimate the expected outcome of selecting a given move. He also discussed learning a model off-line to predict outcome from static features of a node. In an optimization context, Juillé and Pollack [1998] used random tree probing as a value choice heuristic during beam search, although no learning was used.

Bresina [1996] used stochastic probing for scheduling, introducing a fixed *ad hoc* bias favoring children preferred by the node-ordering heuristic. Adaptive probing provides a way to estimate that bias on-line, rather than having to specify it beforehand, presumably using trial and error. By removing this burden from the user, it also becomes feasible to use a more flexible model.

The DTS system of Othar and Hansson [1994] uses learning during search to help allocate effort. Their method learns a function from the value of a heuristic function at a node to the node's probability of being a goal and the expected effort required to explore the node's subtree. It then explores nodes with the greatest expected payoff per unit of effort. In a similar vein, Bedrax-Weiss [1999] proposed weighted discrepancy search, which uses a training set of similar problems to estimate the probability that a node has a goal beneath it, and uses the distribution of these values to derive an optimal searching policy. Adaptive probing is less ambitious and merely estimates action costs rather than goal probability.

Squeaky-wheel optimization [Joslin and Clements, 1998] adapts during tree search, although it learns a variable ordering for use with a greedy constructive algorithm, rather than learning about the single tree that results from using an ordinary variable choice heuristic. The relative benefits of adapting the variable ordering as opposed to the value ordering seem unclear at present. Adaptive probing is slightly more general, as the squeaky-wheel method requires the user to specify a domain-specific analysis function for identifying variables that should receive increased priority during the next probe.

Adaptive tree probing is similar in spirit to iterative improvement algorithms such as adaptive multi-start [Boese *et al.*, 1994], PBIL [Baluja, 1997], and COMIT [Baluja and Davies, 1998] which explicitly try to represent promising regions in the search space and generate new solutions from that representation. For some problems, however, tree search is more natural and heuristic guidance is more easily expressed over extensions of a partial solution in a constructive algorithm than over changes to a complete solution. Adaptive probing gives one the freedom to pursue incomplete heuristic search in whichever space is most suitable for the problem. It is a promising area of future research to see how the two types of heuristic information might be combined.

The Greedy Random Adaptive Search Procedure (GRASP) of Feo and Resende [1995] is, in essence, heuristic-biased stochastic probing with improvement search on each leaf. Adaptive probing provides a principled, relatively parameter-free, way to perform the probing step. Similarly, aspects of Ant Colony Optimization algorithms [Dorigo and Gambardella, 1997], in which 'pheremone' accumulates to represent the information gathered by multiple search trials, can be seen as an approximation of adaptive probing.

Adaptive probing is also related to STAGE [Boyan and Moore, 1998], which attempts to predict promising starting points for hill-climbing given the values of user-specified problem-specific features. The discrepancy cost model requires less of the user, however, since the usual node-ordering function is used as the only problem-specific feature. The tree structure itself can be used to give the geometry for the search space model.

Although adaptive tree probing seems superficially like traditional reinforcement learning, since we are trying to find good actions to yield the best reward, important details differ. Here, we always start in the same state, choose several actions at once, and transition deterministically to a state we have probably never seen before to receive a reward. Rather than learning about sequences of actions through multiple states, our emphasis is on representing the possible action sets compactly to facilitate generalization about the reward of various sets of actions. We assume independence of actions, which collapses the breadth of the tree, and additivity of action costs, which allows learning from leaves. In essence, we generalize over both states and actions.

5 Possible Extensions

The particular adaptive probing algorithm we have evaluated is only one possible way to pursue this general approach. It would be interesting to try more restricted models, perhaps forcing action costs to be a smooth function of depth, for example. It may be worthwhile to distribute variance unequally among depths. Additional features besides depth might be helpful, perhaps characterizing the path taken so far.

The algorithm we have investigated here takes no prior experience into account. An initial bias in favor of the heuristic may be beneficial. Furthermore, it may also be possible to reuse the learned models across multiple problems in the same domain.

Adaptive probing can be used for goal search, as we saw with boolean satisfiability, as long as a maximum depth and a measure of progress are available. If a measure of leaf quality

is not available, it may be possible to fit the model using many small instances of similar problems (or small versions of the current problem) that can be quickly solved and then to scale up the model to guide probing on the original problem.

6 Conclusions

It is a widely held intuition that tree search is only appropriate for complete searches, while local improvement search dominates in hard or poorly understood domains. Adaptive probing can overcome the strong assumptions that are built into systematic tree search procedures. By learning a model of the tree on-line and simultaneously using it to guide search, we have seen how incomplete heuristic search can be effective in a tree-structured search space. When the node-ordering heuristic is very accurate, a systematic discrepancy search algorithm may be more effective. But for problems with unknown character or domains that are less well-understood, the robustness of adaptive probing makes it superior. Its flexibility raises the possibility that, for difficult and messy problems, incomplete tree search may even be a viable alternative to local improvement algorithms.

Acknowledgments

Thanks to Stuart Shieber, Avi Pfeffer, Irvin Schick, Rocco Servedio, Jeff Enos, and the Harvard AI Research Group for their many helpful suggestions and comments. This work was supported in part by NSF grants CDA-94-01024 and IRI-9618848.

References

[Abramson, 1991] Bruce Abramson. *The Expected-Outcome Model of Two-Player Games*. Pitman, 1991.

[Baluja and Davies, 1998] Shumeet Baluja and Scott Davies. Fast probabilistic modeling for combinatorial optimization. In *Proceedings of AAAI-98*, 1998.

[Baluja, 1997] Shumeet Baluja. Genetic algorithms and explicit search statistics. In Michael C. Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems 9*, 1997.

[Bedrax-Weiss, 1999] Tania Bedrax-Weiss. *Optimal Search Protocols*. PhD thesis, University of Oregon, Eugene, OR, August 1999.

[Boese *et al.*, 1994] Kenneth D. Boese, Andrew B. Kahng, and Sudhakar Muddu. A new adaptive multi-start technique for combinatorial global optimizations. *Operations Research Letters*, 16:101–113, 1994.

[Boyan and Moore, 1998] Justin A. Boyan and Andrew W. Moore. Learning evaluation functions for global optimization and boolean satisfiability. In *Proceedings of AAAI-98*, 1998.

[Bresina, 1996] John L. Bresina. Heuristic-biased stochastic sampling. In *Proceedings of AAAI-96*, pp 271–278, 1996.

[Cesa-Bianchi *et al.*, 1996] Nicolò Cesa-Bianchi, Philip M. Long, and Manfred K. Warmuth. Worst-case quadratic loss bounds for on-line prediction of linear functions by gradient descent. *IEEE Transactions on Neural Networks*, 7(2):604–619, 1996.

[Crawford and Baker, 1994] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of AAAI-94*, pages 1092–1097, 1994.

[Dorigo and Gambardella, 1997] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[Feo and Resende, 1995] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[Gent and Walsh, 1996] Ian P. Gent and Toby Walsh. Phase transitions and annealed theories: Number partitioning as a case study. In *Proceedings of ECAI-96*, 1996.

[Hansson and Mayer, 1994] Othar Hansson and Andrew Mayer. Dts: A decision-theoretic scheduler for space telescope applications. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, chapter 13, pages 371–388. Morgan Kaufmann, San Francisco, 1994.

[Harvey and Ginsberg, 1995] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of IJCAI-95*, pages 607–613, 1995.

[Johnson *et al.*, 1991] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; Part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, May-June 1991.

[Joslin and Clements, 1998] David E. Joslin and David P. Clements. "Squeaky wheel" optimization. In *Proceedings of AAAI-98*, pages 340–346. MIT Press, 1998.

[Juillé and Pollack, 1998] Hughes Juillé and Jordan B. Pollack. A sampling-based heuristic for tree search applied to grammar induction. In *Proceedings of AAAI-98*, pages 776–783. MIT Press, 1998.

[Karmarkar and Karp, 1982] Narenda Karmarkar and Richard M. Karp. The differencing method of set partitioning. Technical Report 82/113, Berkeley, 1982.

[Karmarkar *et al.*, 1986] Narenda Karmarkar, Richard M. Karp, George S. Lueker, and Andrew M. Odlyzko. Probabilistic analysis of optimum partitioning. *Journal of Applied Probability*, 23:626–645, 1986.

[Korf, 1995] Richard E. Korf. From approximate to optimal solutions: A case study of number partitioning. In *Proceedings of IJCAI-95*, 1995.

[Korf, 1996] Richard E. Korf. Improved limited discrepancy search. In *Proceedings of AAAI-96*, pp 286–291, 1996.

[Meseguer, 1997] Pedro Meseguer. Interleaved depth-first search. In *Proceedings of IJCAI-97*, pp 1382–1387, 1997.

[Walsh, 1997] Toby Walsh. Depth-bounded discrepancy search. In *Proceedings of IJCAI-97*, 1997.

Heuristic Search in Infinite State Spaces Guided by Lyapunov Analysis

Theodore J. Perkins and **Andrew G. Barto**
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003, USA
{perkins,barto}@cs.umass.edu

Abstract

In infinite state spaces, many standard heuristic search algorithms do not terminate if the problem is unsolvable. Under some conditions, they can fail to terminate even when there are solutions. We show how techniques from control theory, in particular Lyapunov stability analysis, can be employed to prove the existence of solution paths and provide guarantees that search algorithms will find those solutions. We study both optimal search algorithms, such as A*, and suboptimal/real-time search methods. A Lyapunov framework is useful for analyzing infinite-state search problems, and provides guidance for formulating search problems so that they become tractable for heuristic search. We illustrate these ideas with experiments using a simulated robot arm.

1 Introduction

As the boundaries become less distinct between artificial intelligence and fields more reliant on continuous mathematics, such as control engineering, it is being recognized that heuristic search methods can play useful roles when applied to problems with infinite state spaces (e.g., Boone [1997], Davies *et al.* [1998]). However, the theoretical properties of heuristic search algorithms differ greatly depending on whether the state space is finite or infinite.

For finite state space problems, a variety of well-understood algorithms are available to suit different needs. For example, the A* algorithm finds optimal solutions when they exist, and is also "optimally efficient"—no other heuristic search algorithm has better worst-case complexity. Variants, such as IDA*, allow more memory-efficient search at the cost of greater time complexity. Conversely, suboptimal search methods, such as depth-first search or best-first search with an inadmissible heuristic, can often produce some solution, typically suboptimal, more quickly than A* can find an optimal solution [Pearl, 1984; Russell and Norvig, 1995]. The RTA* algorithm is able to choose actions in real-time, while still guaranteeing eventual arrival at a goal state [Korf, 1990]. However, if the state space is infinite, none of these algorithms is guaranteed to have the same properties. A* is complete only if additional conditions hold on the costs of search operators, and it does not terminate if the problem admits no solution [Pearl, 1984]. Suboptimal search methods may not terminate, even if a closed list is maintained. RTA* is not guaranteed to construct a path to a goal state [Korf, 1990].

A further difficulty when infinite state spaces are considered is that the most natural problem formulations often include infinite action spaces. To apply heuristic search, one must select a finite subset of these actions to be explored in any given state. In doing so, the possibility arises that an otherwise reachable goal set becomes unreachable.

Some of these difficulties are unavoidable in general. With an infinite number of possible states, a search problem may encode the workings of a Turing machine, including its memory tape. Thus, the question of whether or not an infinite-state search problem has a solution is in general undecidable. However, it is possible to address these difficulties for useful subclasses of problems. In this paper we examine how Lyapunov analysis methods can help address these difficulties when it is applicable. A Lyapunov analysis of a search problem relies on domain knowledge taking the form of a Lyapunov function. The existence of a Lyapunov function guarantees that some solution to the search problem exists. Further, it can be used to prove that various search algorithms will succeed in finding a solution. We study two search algorithms in detail: A* and a simple iterative, real-time method that incrementally constructs a solution path. Our main goal is to show how Lyapunov methods can help one analyze and/or formulate infinite-state search problems so that standard heuristic search algorithms are applicable and can find solutions.

The paper is organized as follows. In Section 2 we define heuristic search problems. Section 3 covers some basics of Lyapunov theory. Sections 4 and 5 describe how Lyapunov domain knowledge can be applied to prove that a search algorithm will find a solution path. The relationship between Lyapunov functions and heuristic evaluation functions is also discussed. Section 6 contains a demonstration of these ideas on a control problem for a simulated robot arm. Section 7 concludes.

2 State Space Search Problems

Definition 1 *A state space search problem (SSP) is a tuple* $(S, G, s_0, \{O_1, \ldots, O_k\})$*, where:*

- *S is the state set. We allow this to be an arbitrary set.*
- *$G \subset S$ is the set of goal states.*
- *$s_0 \notin G$ is the initial, or start, state.*
- *$\{O_1, \ldots, O_k\}$ is a set of search operators. Some search operators may not be applicable in some states. When a search operator O_j is applied to a state $s \notin G$, it results in a new state $Succ_j(s)$ and incurs a cost $c_j(s) \geq 0$.*

A solution to an SSP is a sequence of search operators that, when applied starting at s_0, results in some state in G. An optimal solution to an SSP is a solution for which the total (summed) cost of the search operators is no greater than the total cost of any other solution.

For infinite state spaces, the infimal cost over all solution paths may not be attained by any path. Thus, solutions may exist without there being any optimal solutions. This possibility is ruled out if there is a universal lower bound $c_{low} > 0$ on the cost incurred by any search operator in any non-goal state [Pearl, 1984]. We will use the phrase "the SSP's costs are bounded above zero" to refer to this property.

Note that our definition of an SSP includes a finite set of search operators $\{O_1, \ldots, O_k\}$, some of which may be unavailable in some states. We have made this choice purely for reasons of notational convenience. The theory we present generalizes immediately to the case in which there is an infinite number of search operators, but the number of operators applicable to any particular non-goal state is finite.

3 Control Lyapunov Functions

Lyapunov methods originated in the study of the stability of systems of differential equations. These methods were borrowed and extended by control theorists. The techniques of Lyapunov analysis are now a fundamental component of control theory and are widely used for the analysis and design of control systems [e.g., Vincent and Grantham 1997].

Lyapunov methods are tools for trying to identify a Lyapunov function for a control problem. In search terms, a Lyapunov function is most easily understood as a descent function (the opposite of a hill-climbing function) with no local minima except at goal states.

Definition 2 *Given an SSP, a control Lyapunov function (CLF) is a function $L : S \mapsto \Re$ with the following properties:*

1. *$L(s) \geq 0$ for all $s \in S$.*
2. *There exists $\delta > 0$ such that for all $s \notin G$ there is some search operator O_j such that $L(s) - L(Succ_j(s)) \geq \delta$.*

The second property asserts that at any non-goal state some search operator leads to a state at least δ down on the CLF. Since a CLF is non-negative, a descent procedure cannot continue indefinitely. Eventually it must reach a state where a δ step down on L is impossible; such a state can only be a goal state.

A CLF is a strong form of domain knowledge, but the benefits of knowing a CLF for a search problem are correspondingly strong. The existence of solutions is guaranteed, and (suboptimal) solutions can be constructed trivially. Numerous sources discuss methods for finding Lyapunov functions (e.g., Vincent and Grantham [1997], Krstić *et al.* [1995]). For many important problems and classes of problems, standard CLFs have already been developed. For example, linear and feedback linearizable systems are easily analyzed by Lyapunov methods. These systems include almost all modern industrial robots [Vincent and Grantham, 1997]. Path planning problems similarly yield to Lyapunov methods [Connolly and Grupen, 1993]. Many other stabilization and control problems, less typically studied in AI, have also been addressed by Lyapunov means, including: attitude control of ships, airplanes, and spacecraft; regulation of electrical circuits and engines; magnetic levitation; stability of networks or queueing systems; and chemical process control (see Levine [1996] for references). Lyapunov methods are relevant to many important and interesting applications.

The definition of a CLF requires that at least one search operator leads down by δ in any non-goal state. A stronger condition is that all search operators descend on the CLF:

Definition 3 *Given an SSP, the set of search operators descends on a CLF L if for any $s \notin G$ there exists at least one applicable search operator, and every applicable search operator O_j satisfies $L(s) - L(Succ_j(s)) \geq \delta$ for some fixed $\delta > 0$.*

4 CLFs and A*

In this section we establish two sets of conditions under which A* is guaranteed to find an optimal solution path in an SSP with an infinite state space. We then discuss several other search algorithms, and the relationship of CLFs to heuristic evaluation functions.

Theorem 1 *If there exists a CLF L for a given SSP, and either of the following conditions are true:*

1. *the set of search operators descends on L, or*
2. *the SSP's costs are bounded above zero,*

then A search will terminate, finding an optimal solution path from s_0 to G.*

Proof: Under condition 1, there are only a finite number of non-goal states reachable from a given start state s_0. The only way for an infinite number of states to be reachable is for them to occur at arbitrarily large depths in the search tree. But since every search operator results in a state at least δ lower on L and $L \geq 0$ everywhere, no state can be at a depth greater than $\lceil L(s_0)/\delta \rceil$. Since the CLF implies the existence of at least one solution and there are only a finite number of reachable states, it follows (e.g., from Pearl [1984], section 3.1), that A* must terminate and return an optimal solution path.

Alternatively, suppose conditions 2 holds. The CLF ensures the existence of at least one solution path. Let this path have cost C. Since each search operator incurs at least c_{low} cost, the f-value of a node at depth d is at least $d \cdot c_{low}$. A* will not expand a node with f-value higher than C. Thus, no node at depth $d > \lceil C/c_{low} \rceil$ will ever be expanded. This means A* must terminate and return an optimal solution. QED.

These results extend immediately to variants of A* such as uniform cost search or IDA*. Under condition 1, not only is

the number of reachable states finite, but these states form an acyclic directed graph and all "leaves" of the graph are goal states. Thus, a great many search algorithms can safely be applied in this case. Under condition 2, depth-first branch-and-bound search is also guaranteed to terminate and return an optimal solution, if it is initialized with the bounding cost C. This follows from nearly the same reasoning as the proof for A*.

How do CLFs relate to heuristic evaluation functions? In general, a CLF seems a good candidate for a heuristic function. It has the very nice property that it can be strictly monotonically decreased as one approaches the goal. Contrast this, for example, to the Manhattan distance heuristic for the 8-puzzle [Russell and Norvig, 1995]. An 8-puzzle solution path typically goes up and down on the Manhattan distance heuristic a number of times, even though the ultimate effect is to get the heuristic down to a value of zero.

However, a CLF might easily fail to be admissible, overestimating the optimal cost-to-goal from some non-goal state. Notice that the definition of a CLF does not depend on the costs of search operators of an SSP. It depends only on S, G, and the successor function. CLFs capture information about the connectivity of the state space more than the costs of search operators.

A possible "fix" to the inadmissibility of a CLF is to scale it. If a CLF is multiplied by any positive scalar, defining a new function $L' = \alpha L$, then L' is also a CLF. If a given CLF overestimates the cost-to-goal for an SSP, then it is possible that a scaled-down CLF would not overestimate. We will not elaborate on the possibility of scaling CLFs to create admissible heuristics in this paper. We will, however, use the scalability of a CLF in the next section, where we discuss real-time search.

5 Real-Time Search

Lyapunov domain knowledge is especially well suited to real-time search applications. As mentioned before, one can rapidly generate a solution path by a simple descent procedure. A generalization of this would be to construct a path by performing a depth-limited search at each step to select the best search operator to apply next. "Best" would mean the search operator leading to a leaf for which the cost-so-far plus a heuristic evaluation is lowest. We will call this algorithm "repeated fixed-depth search" or RFDS.

Korf [1990] described this procedure in a paper on real-time search and, rightly, dismissed it because in general it is not guaranteed to produce a path to a goal state, even in finite state spaces. However, the procedure is appealing in its simplicity and appropriate for real-time search in that the search depth can be set to respond to deadlines for the selection of search operators. In this section we examine conditions under which this procedure can be guaranteed to construct a path to a goal state.

Theorem 2 *Given an SSP and a CLF, L, for that SSP, if the set of search operators descends on L then RFDS with any depth limit $d \geq 1$ and any heuristic evaluation function will terminate, constructing a complete path from s_0 to G.*

Proof: At each step, any search operator that is appended to the growing solution will cause a step down on L of at least δ, which inevitably leads to G. QED.

A more interesting and difficult case is when some of the operators may not be descending, so that the solution path may travel up and down on the CLF.

Theorem 3 *Given an SSP whose costs are bounded above zero and a CLF, L. Suppose that L is used as a heuristic evaluation of non-goal leaves, and that $L(s) - L(Succ_1(s)) \geq c_1(s)$ for all $s \notin G$. Then RFDS with any depth limit $d \geq 1$ will terminate, constructing a complete path from s_0 to G.*

Note that the theorem assumes that O_1 has the special property $L(s) - L(\mathrm{Succ}_1(s)) \geq c_1(s)$. More generally, in each state there must be some search operator that satisfies this property. It is only for notational convenience that we have assumed that O_1 satisfies the property everywhere.

To prove Theorem 3, we require some definitions. Suppose RFDS generates a path $s_0, s_1, \ldots, s_N$, where s_N may or may not be a goal state but the other states are definitely non-goal. At the t^{th} step, $t \in \{0, \ldots, N\}$, RFDS generates a search tree to evaluate the best operator to select next. Let g_t be the cost of the search operators leading up to state s_t; let l_t be the leaf-state with the best (lowest) evaluation in the t^{th} search tree; let $c_{t,1}, \ldots, c_{t,n(t)}$ be the costs of the search operators from s_t to l_t; and let h_t be the heuristic evaluation of l_t—zero if the leaf corresponds to a goal state, and $L(l_t)$ otherwise. Define $f_t = g_t + \sum_{i=1}^{n(t)} c_{t,i} + h_t$.

Lemma 1 *Under the conditions of Theorem 3, $f_t \geq f_{t+1}$ for $t \in \{0, \ldots, N-1\}$.*

Proof: $f_t = g_t + \sum_{i=1}^{n(t)} c_{t,i} + h_t = g_{t+1} + \sum_{i=2}^{n(t)} c_{t,i} + h_t$ because s_{t+1} is one step along the path to the best leaf of iteration t. Thus, the inequality we need to establish is $\sum_{i=2}^{n(t)} c_{t,i} + h_t \geq \sum_{i=1}^{n(t+1)} c_{t+1,i} + h_{t+1}$. The right side of this inequality is simply the cost of some path from s_{t+1} to l_{t+1} plus a heuristic evaluation. The left side corresponds to a path from s_{t+1} to l_t, plus a heuristic evaluation.

First, suppose l_t is a goal state. The path from s_{t+1} to l_t will be included among those over which RFDS minimizes at the t^{th} step. Thus, in this case the inequality holds.

If l_t is not a goal state, then the search at iteration $t+1$ expands l_t. The evaluation of the path from s_{t+1} to $\mathrm{Succ}_1(l_t)$ is

$$\begin{aligned} & \textstyle\sum_{i=2}^{n(t)} c_{t,i} + c_1(l_t) + h(\mathrm{Succ}_1(l_t)) \\ \leq\ & \textstyle\sum_{i=2}^{n(t)} c_{t,i} + c_1(l_t) + L(\mathrm{Succ}_1(l_t)) \\ \leq\ & \textstyle\sum_{i=2}^{n(t)} c_{t,i} + L(l_t) \\ =\ & \textstyle\sum_{i=2}^{n(t)} c_{t,i} + h_t. \end{aligned}$$

Thus, this path's cost meets the inequality, and since RFDS minimizes over that and other paths, the inequality must hold for the best path of iteration $t+1$. QED.

Proof of Theorem 3: Suppose RFDS runs forever without constructing a path to a goal state. From the previous lemma, $f_t \leq f_0$ for all $t \geq 0$. Since the SSP's costs are bounded above zero, $f_t \geq g_t \geq t \cdot c_{low}$. For sufficiently large t, this contradicts $f_t \leq f_0$. Thus RFDS does construct a path to goal. QED.

Theorem 3 requires a relationship between the decrease in a CLF caused by application of O_1 and the cost incurred

by O_1. A given CLF may not satisfy this property, or one may not know whether the CLF satisfies the property or not. We propose two methods for generating a CLF guaranteed to have the property, assuming that applying O_1 to any non-goal state causes a decrease in the original CLF of at least δ, for some fixed $\delta > 0$.

The first method is scaling, which is applicable when there is a maximum cost c_{high} that operator O_1 may incur when applied to any non-goal state. In this case, consider the new CLF $L' = \frac{c_{high}}{\delta} L$. For any $s \notin G$, $L'(s) - L'(\text{Succ}_1(s)) = (c_{high}/\delta) \cdot (L(s) - L(\text{Succ}_1(s))) \geq (c_{high}/\delta) \cdot \delta = c_{high} \geq c_1(s)$. Thus, L' is a CLF satisfying the conditions of Theorem 3.

A problem with this scheme is that one or both of the constants c_{high} and δ_1 may be unknown. An overestimate of the constant $\frac{c_{high}}{\delta_1}$ will produce a new CLF satisfying the theorem. If an overestimate cannot be made directly, then a scaling factor can be determined on-line, as RFDS runs. Suppose one defines $L' = \alpha L$ for any initial guess α. One can than perform RFDS, checking the condition $L'(s) - L'(\text{Succ}_1(s)) \geq c_1(s)$ every time O_1 is applied to some state. If the condition is violated, update the scaling factor, e.g., as $\alpha \leftarrow \frac{c_1(s)}{L'(s) - L'(\text{Succ}_1(s))} + \varepsilon$, for some fixed $\varepsilon > 0$. This update rule ensures that the guess α is consistent with all the data observed so far and will meet or exceed $\frac{c_{high}}{\delta_1}$ in some finite number of updates. If RFDS does not construct a path to a goal state by the time that a sufficient number of updates are performed, then Theorem 3 guarantees the completion of the path afterward.

A second method for generating a CLF that meets the conditions of Theorem 3 is to perform roll-outs [Tesauro and Galperin, 1996; Bertsekas *et al.*, 1997]. In the present context, this means defining $L'(s)$ = "the cost of the solution path generated by repeated application of O_1 starting from s and until it reaches G". The function L' is evaluated by actually constructing the path. The reader may verify that L' meets the definition of a CLF and satisfies the requirements of Theorem 3. Performing roll-outs can be an expensive method for evaluating leaves because an entire path to a goal state needs to be constructed for each evaluation. However, roll-outs have been found to be quite effective in both game playing and sequential control [Tesauro and Galperin, 1996; Bertsekas *et al.*, 1997].

6 Robot Arm Example

We briefly illustrate the theory presented above by applying it to a problem requiring the control of the simulated 3-link robot arm, depicted in Figure 1. The state space of the arm is $\Re^6$, corresponding to three angular joint positions and three angular joint velocities. We denote the joint position and joint velocity column vectors by θ and $\dot{\theta}$. The set G of goal states is $\{s \in S : \|s\| \leq 0.01\}$, which is a small hyper-rectangle of states in which the arm is nearly stationary in the straight-out horizontal configuration.

The dynamics of mechanical systems such as a robot arm are most naturally described in continuous time. A standard

link	length	mass
1	0.1	1.0
2	1.0	1.0
3	1.0	1.0

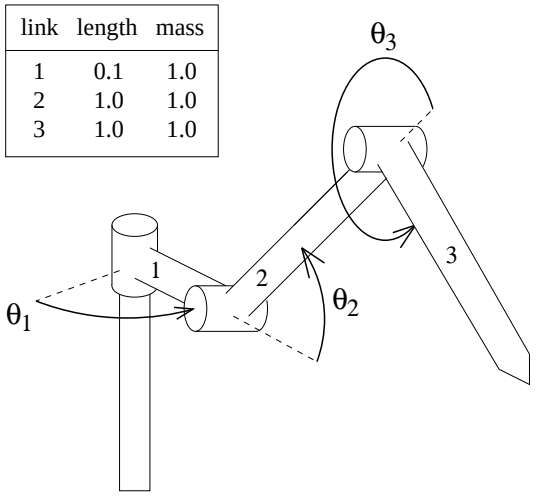

Figure 1: Diagram of the 3-Link Robot Arm

model for a robot arm is [Craig, 1989]:

$$\frac{\mathrm{d}}{\mathrm{dt}} s = \frac{\mathrm{d}}{\mathrm{dt}} \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta} \\ H^{-1}(\theta, \dot{\theta})(\tau - V(\theta, \dot{\theta}) - g(\theta)) \end{bmatrix}$$

where g represents gravitational forces, V represents Coriolis and other velocity-dependent forces, H is the inertia matrix, and τ is the actuator torques applied at the joints.

We develop a set of continuous time controllers—rules for choosing τ—-based on feedback linearization and linear-quadratic regulator (LQR) methods [e.g., Vincent and Grantham, 1997]. These controllers form the basis of the search operators described in the next section. Feedback linearization amounts to reparameterizing the control torque in terms of a vector u, where $\tau = Hu + V + g$. This puts the robot dynamics into the particularly simple linear form

$$\frac{\mathrm{d}}{\mathrm{dt}} \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta} \\ u \end{bmatrix}$$

to which LQR control design is applicable. An LQR controller is a rule for choosing u, and thus τ, based on θ and $\dot{\theta}$. LQR design yields two main benefits: 1) a simple controller that can asymptotically bring the arm to any specified target state, 2) a Lyapunov function that can be used as a CLF for the problem of getting to a goal state. We define five basic controllers:

- C_1: An LQR controller with $\theta = (0,0,0)$ as the target configuration. This configuration is within G, so C_1 alone is able to bring the arm into G from any initial position. We use the associated Lyapunov function as a CLF for the SSP we are constructing: $L_{\text{arm}} = s^T Q s$, where Q is a symmetric positive definite matrix produced during the standard computations for the LQR design.
- C_2: Chooses τ as an LQR controller with target configuration $\theta = (0,0,0)$ except that u is multiplied by 2. This tends to cause higher control torques that bring the arm towards the target configuration faster.

- C_3: Similar to C_2, but u is multiplied by 0.5 instead of 2.
- C_4: An LQR controller that does not apply any torque to the first joint, but regulates the second two joints to configuration $(\theta_2, \theta_3) = (\pi/4, -\pi/2)$. This tends to fold the arm in towards the center, so it can swing around the first joint more easily.
- C_5: Similar to C_4, but regulating the second two joints to the configuration $(\theta_2, \theta_3) = (\pi/2, -\pi)$.

6.1 Search Operators

We construct two sets of five search operators. In the first set of search operators, Ops_1, each operator corresponds to one of the five controllers above. $\text{Succ}_j(s)$ is defined as the state that would result if the arm started in state s and C_j controlled the arm for $\Delta = 0.25$ time interval.

We define the cost of applying a search operator to be the continuous time integral of the quantity $\|\theta(t)\|^2 + \|\tau(t) - \tau_0\|^2$, where $\theta(t)$ are the angles the arm goes through during the Δ-time interval, $\tau(t)$ are the joint torques applied, and τ_0 is the torque needed to hold the arm steady, against gravity, at configuration $\theta = (0,0,0)$. This kind of performance metric is standard in control engineering and robotics. The term $\|\theta(t)\|^2$ reflects the desire to move the arm to G; the term $\|\tau(t) - \tau_0\|^2$ penalizes the use of control torques to the extent that they differ from τ_0. Defined this way, the costs of all search operators are bounded above zero.

The second set of search operators, Ops_2, is also based on controllers C_1 through C_5, but the continuous-time execution is somewhat different. When a controller $C_j,\ j \in \{2,\ldots,5\}$ is being applied for a Δ-time interval, the time derivative of L_{arm}, $\dot{L}_{\text{arm}}$, is constantly computed. If $\dot{L}_{\text{arm}}$ is greater than -0.1, the torque choice of C_1 is supplied instead. Under C_1, $\dot{L}_{\text{arm}} \leq \delta_1 < 0$ for some unknown δ_1. In this way, $\dot{L}_{\text{arm}}$ is bounded below zero for all the Ops_2 search operators, and thus they all step down L_{arm} by at least the (unknown) amount $\delta = \Delta \cdot \min(0.1, \delta_1)$ with each application. The costs for applying these search operators is defined in the same way as for Ops_1.

6.2 Experiments and Results

We tested A* and RFDS on the arm with both sets of search operators, averaging results over a set of nine initial configurations $\theta_0^T = (x, y, -y)$ for $x \in \{-\pi, -2\pi/3, -\pi/3\}$ and $y \in \{-\pi/2, 0, +\pi/2\}$. Theorem 1 guaranteed that A* would find an optimal solution within either set of search operators. We ran RFDS at a variety of depths and with three different heuristic leaf evaluation functions: zero (RFDS-Z), roll-outs (RFDS-R), and a scaled version of L_{arm} (RFDS-S). Because we did not know an appropriate scaling factor for L_{arm}, for each starting configuration we initialized the scaling factor to zero and updated it as discussed in Section 5, using $\varepsilon = 0.01$. All of the RFDS runs under Ops_2 were guaranteed to generate solutions by Theorem 2. For the RFDS-R and RFDS-S runs with Ops_1, Theorem 3 provided the guarantee of producing solutions.

The results are shown in Figures 2 and 3, which report average solution cost, nodes expanded by the search, and the amount of virtual time for which the robot arm dynamics were simulated during the search. Simulated time is closely related to the number of nodes expanded, except that the RFDS-R figures also account for the time spent simulating the roll-out path to the goal. The actual CPU times used were more than an order of magnitude smaller than the simulated times, and our code has not been optimized for speed. The "C_1 only" row gives the average solution cost produced by controlling the arm using C_1 from every starting point. Achieving this level of performance requires no search at all. A* with either set of search operators found significantly lower cost solutions than those produced by C_1. The A* solution qualities with Ops_1 and Ops_2 are very close. We anticipated that Ops_2 results might not be as good, because the paths are constrained to always descend on L_{arm}, whereas Ops_1 allows the freedom of ascending. For this problem, the effect is small. A* with Ops_2 involved significantly less search effort than with Ops_1, in terms of both the number of expanded nodes and the amount of arm simulation time required.

The RFDS-Z – Ops_1 combination is the only one not guaranteed to produce a solution, and indeed, it did not. Under Ops_2, RFDS-Z is guaranteed to produce a path to a goal for any search depth. At low search depths, the solutions it produced were worse than those produced by C_1. However, at greater depth, it found significantly better solutions, and with much less search effort than required by A*.

RFDS-R with either set of search operators produced excellent solutions at depth one. This result is quite surprising. Apparently the roll-out yields excellent information for distinguishing good search operators from bad ones. Another unexpected result is that greater search depths did not improve over the depth-one performance. Solution quality is slightly worse and search effort increased dramatically. At the higher search depths, RFDS-R required more arm simulation time than did A*. This primarily resulted from the heuristic leaf evaluations, each of which required the simulation of an entire trajectory to G.

RFDS-S produced good solutions with very little search effort, but greater search effort did not yield as good solutions as other algorithms were able to produce. RFDS-S seems to be the most appropriate algorithm for real-time search if very fast responses are needed. It produced solutions that significantly improved over those produced by C_1, using search effort that was less than one tenth of that of the shallowest roll-out runs, and less than one hundredth of the effort required by A*.

6.3 Discussion

The arm simulation results reflect some of the basic theory and expectations developed earlier in the paper. The algorithms that were guaranteed to find solutions did, and those that were not guaranteed to find solutions did not. These results depend on, among other things, our choice of the "duration" of a search operator, Δ. We ran the same set of experiments for four other choices of Δ: 1.0, 0.75, 0.5, and 0.1. Many of the qualitative results were similar, including the excellent performance of roll-outs with a depth-one search.

One difference was that for higher Δ, A* became more competitive with RFDS in terms of search effort. Con-

Algorithm	Sol'n Cost	Nodes Expanded	Sim. Time
C_1 only	435.0	0	0
A*	304.0	11757.4	14690.7
RFDS-Z depths 1-5	∞	∞	∞
RFDS-R depth 1	305.7	41.3	667.6
RFDS-R depth 2	307.1	158.6	2001.1
RFDS-R depth 3	307.1	477.9	5489.8
RFDS-R depth 4	307.1	1299.0	13214.6
RFDS-R depth 5	307.1	3199.6	27785.5
RFDS-S depth 1	354.3	32.9	49.0
RFDS-S depth 2	371.0	163.2	210.7
RFDS-S depth 3	359.4	675.3	848.8
RFDS-S depth 4	343.4	2669.4	3337.6
RFDS-S depth 5	334.4	9565.3	11952.1

Figure 2: Ops_1 Results

Algorithm	Sol'n Cost	Nodes Expanded	Sim. Time
C_1 only	435.0	0	0
A*	305.0	4223.3	3492.8
RFDS-Z depth 1	502.8	45.0	40.6
RFDS-Z depth 2	447.8	113.2	94.7
RFDS-Z depth 3	385.2	226.8	192.9
RFDS-Z depth 4	360.9	434.9	374.0
RFDS-Z depth 5	326.7	743.6	648.0
RFDS-R depth 1	306.4	38.2	479.8
RFDS-R depth 2	307.8	118.7	1424.1
RFDS-R depth 3	307.8	333.1	4047.0
RFDS-R depth 4	307.8	920.7	10085.3
RFDS-R depth 5	307.8	2359.6	21821.6
RFDS-S depth 1	355.3	33.0	28.2
RFDS-S depth 2	370.7	105.1	87.9
RFDS-S depth 3	359.2	324.3	284.3
RFDS-S depth 4	342.9	1019.6	920.9
RFDS-S depth 5	330.3	3011.4	2758.3

Figure 3: Ops_2 Results

versely, at $\Delta = 0.1$, A* exhausted the computer's memory and crashed. The complexity of A* generally grows exponentially with the length of the optimal solution. For RFDS, which iteratively constructs a solution, effort is linear in the length of the solution it produces. The complexity of RFDS is more governed by the search depth, which can be chosen independently. Another difference observed at $\Delta = 1.0$ is that RFDS-Z succeeded in producing solutions from all starting positions when the search depth was four or higher.

For all algorithms, solution quality was lower for higher Δ. This is simply because lower Δ allows finer-grained control over the trajectory that the arm takes. The freedom to choose Δ suggests an alternative means for performing real-time search: one could perform a sequence of A* searches, initially with a high Δ and then decrease Δ to find solutions at finer temporal resolutions.

7 Conclusion

We demonstrated how domain knowledge in the form of a Lyapunov function can help one analyze and formulate infinite-state search problems. Lyapunov methods guarantee the existence of solutions to a problem, and they can be used to show that both optimal and suboptimal/real-time search procedures will find those solutions. These results provide a theoretical basis for extending the range of problems to which heuristic search methods can be applied with a guarantee of results.

Acknowledgments

This work was funded by the National Science Foundation under Grant No. ECS-0070102. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Theodore Perkins is also supported by a graduate fellowship from the University of Massachusetts Amherst.

References

[Bertsekas *et al.*, 1997] D. P. Bertsekas, J. N. Tsitsiklis, and C. Wu. Rollout algorithms for combinatorial optimization. *Journal of Heuristics*, 1997.

[Boone, 1997] G. Boone. Minimum-time control of the acrobot. In *1997 International Conference on Robotics and Automation*, pages 3281–3287, 1997.

[Connolly and Grupen, 1993] C. I. Connolly and R. A. Grupen. The applications of harmonic functions to robotics. *Journal of Robotics Systems*, 10(7):931–946, 1993.

[Craig, 1989] J. J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley, 1989.

[Davies *et al.*, 1998] S. Davies, A. Ng, and A. Moore. Applying online search techniques to continuous-state reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 753–760, 1998.

[Korf, 1990] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, (42):189–211, 1990.

[Krstić *et al.*, 1995] M. Krstić, I. Kanellakopoulos, and P. Kokotović. *Nonlinear and Adaptive Control Design*. John Wiley & Sons, Inc., New York, 1995.

[Levine, 1996] W. S. Levine, editor. *The Control Handbook*. CRC Press, Inc., 1996.

[Pearl, 1984] J. Pearl. *Heuristics*. Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1984.

[Russell and Norvig, 1995] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1995.

[Tesauro and Galperin, 1996] G. Tesauro and G. R. Galperin. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing: Proceedings of the Ninth Conference*. MIT Press, 1996.

[Vincent and Grantham, 1997] T. L. Vincent and W. J. Grantham. *Nonlinear and Optimal Control Systems*. John Wiley & Sons, Inc., New York, 1997.

A backbone-search heuristic for efficient solving of hard 3-SAT formulae*

Olivier Dubois† and **Gilles Dequen**‡
† LIP6, C.N.R.S.-Université Paris 6, 4 place Jussieu, 75252 Paris cedex 05, France.
‡ LaRIA, Université de Picardie, Jules Verne, C.U.R.I., 5 Rue du moulin neuf, 80000 Amiens, France.
e-mail : Olivier.Dubois@lip6.fr, dequen@laria.u-picardie.fr

Abstract

Of late, new insight into the study of random k-SAT formulae has been gained from the introduction of a concept inspired by models of physics, the 'backbone' of a SAT formula which corresponds to the variables having a fixed truth value in all assignments satisfying the maximum number of clauses. In the present paper, we show that this concept, already invaluable from a theoretical viewpoint in the study of the satisfiability transition, can also play an important role in the design of efficient DPL-type algorithms for solving hard random k-SAT formulae and more specifically 3-SAT formulae. We define a heuristic search for variables belonging to the backbone of a 3-SAT formula which are chosen as branch nodes for the tree developed by a DPL-type procedure. We give in addition a simple technique to magnify the effect of the heuristic. Implementation yields DPL-type algorithms with a significant performance improvement over the best current algorithms, making it possible to handle unsatisfiable hard 3-SAT formulae up to 700 variables.

1 Introduction

The Satisfiability Problem, as one of the basic NP-complete problems [Garey and Johnson, 1979], has appeared to possess a property of great interest with respect to computational complexity. Random CNF formulae in which each clause has a fixed number k of literals, known as k-SAT formulae, exhibit with respect to satisfiability, a so-called phase transition phenomenon such that the SAT formulae at the transition appear in probability the most difficult to solve. This property, initially described in [Mitchell *et al.*, 1992] and since studied in numerous papers [Crawford and Auton, 1993; Gent and Walsh, 1994] has the following practical manifestation. When in an experiment m clauses with exactly k literals, $k = 2, 3, 4$ ... (limited to manageable sizes of formulae), over a set of n boolean variables, are chosen at random with a fixed ratio $c = m/n$, the probability of satisfiability falls abruptly from near 1 to near 0 as c passes some critical value called the threshold. Moreover (leaving aside 2-SAT formulae which are well known to be solvable in polynomial time), at the threshold, for $k \geq 3$ a peak of difficulty of solving is observed and this peak grows exponentially as a function of n. This intriguing property correlating the satisfiability threshold with the hardness of k-SAT formulae, has stimulated many theoretical as well as experimental studies, eliciting a better understanding of the phase transition phenomenon and progress in solving hard SAT formulae.

Recently, the satisfiability phase transition has attracted much attention from physicists. Sophisticated models of statistical physics have been evolved over the past few decades to predict and analyze certain phase transitions, and these can be compared, from a theoretical viewpoint, with the satisfiability transition. This has shed new light on the question. In particular in 1999, an insightful concept was introduced, the 'backbone' of a SAT formula [Monasson *et al.*, 1999]. The backbone corresponds to the set of variables having a fixed truth value in all assignments satisfying the maximum number of clauses of a k-SAT formula (cf MaxSAT). The relevance of this concept is connected to the fact that the number of solutions (i.e. satisfying assignments) of a satisfiable random k-SAT formula has been proved to be almost surely exponential as a function of n up to and at the threshold [Boufkhad and Dubois, 1999], a result corroborated in its MaxSAT version by the physics model known as 'Replica Symmetric ansatz'. This concept of backbone of a k-SAT formula has turned out to play an important role in theoretical studies, allowing for example, the scaling window of the 2-SAT transition to be determined [Bollobás *et al.*, 2000]. We show in this paper that the concept of backbone can also play an important role in more efficiently solving hard k-SAT formulae. We will focus particularly on 3-SAT formulae which are the easiest k-SAT formulae to handle (for $k \geq 3$) and therefore at present the most studied in the literature. The hard 3-SAT formulae are generated randomly with a ratio c around 4.25 which appears experimentally to be close to the threshold value. In the last ten years, the best-performing complete solving algorithms (i.e., those definitely determining whether a solution exists or not) have been based on the classical DPL procedure [Davis *et al.*, 1962]. Important progress was achieved, making it possible at present, for example, to solve 3-SAT hard formulae with 300 variables (and therefore with 1275 clauses) in roughly the same computing time as formulae with 100 variables (and 425 clauses) ten years ago under equivalent com-

*This work was supported by Advanced Micro Devices Inc.

puting conditions.
The DPL-type procedures which have led to this progress were all designed according to a single viewpoint, that of dealing with the clauses which remain not satisfied at successive nodes of the solving tree, trying mainly to reduce them as much as possible in number and in size. Clauses already satisfied in the course of solving were not considered. By using the concept of backbone we change this viewpoint, focusing on the clauses which *can* be satisfied in the course of solving, and *how*. Thus, we first explain in this paper how with this new viewpoint, the size of trees developed by a DPL-type procedure can be efficiently reduced. Then we give practically a simple branching heuristic based on the search for variables belonging to the backbone of a formula, and we show that it brings immediately an improvement in performance compared with the best current heuristic, this improvement increasing with the number of variables. We further present a technique to magnify the effect of the heuristic. Combining this heuristic and this technique in a DPL-type procedure, we obtain significant performance improvements over existing SAT solvers in the literature, solving unsatisfiable hard random 3-SAT formulae with 200 to 600 variables from 1.2 to about 3 times faster than by the current best suitable algorithms, particularly *satz214* [Li, 1999]. Moreover, hard 3-SAT formulae up to 700 variables are shown to be solvable by our procedure in about 25 days of computing time, which begins to be relatively acceptable. These experimental results reported in the present paper bring an up beat view in contrast to the misgivings expressed in [Li and Gerard, 2000] about the possibility to obtain further significant progress with DPL-type algorithms, which went as far as to suggest that the best current algorithms could be close to their limitations.

2 An approach using the concept of backbone for designing efficient DPL-type procedures.

In this section, we need to generalize the notion of backbone of a SAT formula F. Given a set of clauses A, no longer necessarily of maximum size, we define the *backbone associated to* A as the set of literals of F having the truth value TRUE in all assignments satisfying the clauses of A.
Since the overall performance of a complete algorithm on random SAT formulae depends essentially on how it handles unsatisfiable formulae, in the remainder of this section we consider only *unsatisfiable* 3-SAT formulae. Let, then, F be an unsatisfiable 3-SAT formula with clauses built on a set L of $2n$ literals, itself derived from a set V of n boolean variables. Let T be the so-called *refutation tree* of F, as developed by a DPL-type procedure. At each leaf ℓ of T, an inconsistency is detected. Let $M(\ell)$ be the subset of clauses of F satisfied by the $\mu(\ell)$ literals set to TRUE along the path $P(\ell)$ from the root of T to the leaf ℓ. Call $S(\ell)$ the set of assignments satisfying $M(\ell)$. Suppose there is a backbone $B(\ell)$ associated to $M(\ell)$. From the above-mentioned theoretical study, this can be expected in practice often to be the case at the leaves of T. The inconsistency detected at ℓ means that $2^{n-\mu(\ell)}$ assignments are refuted as possible solutions of F. We have the inequality: $2^{n-\mu(\ell)} \leq |S(\ell)|$. Equality obtains precisely when the set of the $\mu(\ell)$ literals constitutes the backbone $B(\ell)$ of $M(\ell)$. Furthermore, the assignments refuted as possible solutions of F at different leaves of T are distinct (form disjoint sets), and their total number equals 2^n. Hence, in lowering the number of leaves of T two factors intervene, namely: *(i)* the size of the sets $S(\ell)$ must be as large as possible, and *(ii)* the equality $2^{n-\mu(\ell)} = |S(\ell)|$ must hold most of the time, at least approximately.
The above considerations extend to all nodes of T. In this case, at any node j, $2^{n-\mu(j)}$ represents the potential maximum number of assignments that can be refuted as possible solutions of F at the node j. According to the above viewpoint, the global condition in order best to reduce the size of T is that, from the root to a nearest node j, literals be chosen which constitute a backbone relative to some subset of clauses of F. Indeed, the nearer the node j is to the root, the smaller the set $M(j)$ of satisfied clauses will tend to be. Now, the crucial point is that any assignment giving the value FALSE to at least one literal of the backbone, can be refuted on the basis of the subset $M(j)$ of clauses associated to $B(j)$. If the set $M(j)$ is of small size, it is then to be expected that any refutation derived from $M(j)$ be short, i.e. that the refutation tree for all such assignments be of small size. Let us illustrate this with the following very simple example. Suppose that among all the clauses of F there are 4 of the form: $\{(x \vee y \vee z); (x \vee y \vee \overline{z}); (x \vee \overline{y} \vee t); (x \vee \overline{y} \vee \overline{t})\}$. The backbone of these 4 clauses reduces to the literal x. According to the above principle, this literal is a preferential choice for a branch stemming from the root. In the branch of the literal $\overline{x}$, refutations are obtained by branching just twice, applying the same principle of bringing forward a backbone at the nearest possible node, thus either y or $\overline{y}$ will be chosen indifferently. Note that in this example the backbone search principle includes in its general form the notion of resolvent without using it. The literal x could of course be inferred by applying classical resolution.
We next detail a heuristic and a technique which embody the foregoing, if only partially.

2.1 A backbone-variables search heuristic

Consider a generic node j of the refutation tree T of F, and let $F(j)$ be the reduced formula at node j. The heuristic (Figure 1) described below aims at selecting literals that may belong to a hypothetical backbone associated to a subset of clauses of F of the smallest possible size. The simple idea sustaining this heuristic is to estimate the number of 'possibilities' for a given literal t of $F(j)$ to be constrained to TRUE in the clauses of $F(j)$ where it appears. E.g., if $F(j)$ contains the clause $(t \vee u \vee v)$, one possibility for t to be TRUE is $(u \vee v) \equiv FALSE$. If in addition $\overline{u}$ and $\overline{v}$ appear p and q times, respectively, in clauses of $F(j)$, the number of possibilities for t to be TRUE can be estimated as pq at this second level by setting to FALSE the necessary literals in the clauses where $\overline{u}$ and $\overline{v}$ appear. If however $\overline{u}$ and $\overline{v}$ appear in both binary and ternary clauses, we must consider the possibility to be greater when a binary rather than a ternary clause is involved, since a single literal is required to have the value FALSE instead of two. To evaluate a 'possibility', we therefore weight the relevant binary and ternary clauses. But for

Set $i \leftarrow 0$ and let *MAX* be an integer.
Proc $h(t)$
$i \leftarrow i + 1$
compute $\mathcal{I}(t)$
if $i <$ *MAX* **then**

$$\text{Return} \sum_{(u \vee v) \in \mathcal{I}(t)} h(\overline{u})h(\overline{v})$$

else

$$\text{Return} \sum_{(u \vee v) \in \mathcal{I}(t)} (2p_1(\overline{u}) + p_2(\overline{u}))(2p_1(\overline{v}) + p_2(\overline{v}))$$

end if

Figure 1: backbone search heuristic *h*

the heuristic evaluation, only the ratio of the weightings matters. As a first apporximation, we take the ratio of probabilities in a random assignment, i.e. simply a ratio of 2. Within the heuristic function, an *evaluation level* is defined for t, as exemplified by the above allusion to a second-level evaluation of t. To be precise, the evaluation level is the number of ternary clauses which must successively be brought into play in order to reach the conclusion that t is TRUE, following the aforementioned principle. Fixing an evaluation level thus limits the number of ternary clauses to be used before concluding that t is TRUE. On the other hand, there is no need to limit the number of *binary* clauses that can be used, with or without ternary ones. Since binary clauses, due to the fact that a single literal has to be FALSE, do not give rise to combinations, their use does not increase the computational complexity of the heuristic. At a given node of the refutation tree, all literals must be evaluated to the same level so as to guarantee comparable evaluations. The heuristic function h in Figure 1 embodies the simple principles just stated. The estimation level equals the number of recursive calls to $h(t)$, namely the constant MAX augmented by 1. For the formal definition of the heuristic, there remains to introduce two types of sets of clauses. $\mathcal{C}(t)$ denotes the set of binary or unary clauses derived by eliminating t, from the clauses of $F(j)$ where t appears. Let $p_1(u)$ and $p_2(u)$ be the number of unary and binary clauses respectively in $\mathcal{C}(u)$. $\mathcal{I}(t)$ is the set of all binary subclauses derived from ternary clauses of $F(j)$ such that each of these binary subclauses set to FALSE, implies t TRUE either directly or by virtue of certain unary clauses having the value FALSE. Let us take an example showing how to compute the heuristic h. Consider the following set of clauses:
$(x \vee \overline{a} \vee b) \wedge (x \vee c) \wedge (\overline{c} \vee d \vee \overline{e}) \wedge (a \vee f \vee \overline{d}) \wedge (a \vee g) \wedge (\overline{b} \vee f \vee g) \wedge (\overline{d} \vee e)$.
We evaluate $h(x)$ on the above set of clauses with *MAX* set to 1. We have $\mathcal{I}(x) = \{\overline{a} \vee b, d \vee \overline{e}\}$. To carry out the evaluation, the sets $\mathcal{C}(\overline{u})$ must be determined for each literal u appearing in the clauses of $\mathcal{I}(x)$, in order to know the values $p_1(\overline{u})$ and $p_2(\overline{u})$. We thus have: $\mathcal{C}(a) = \{f \vee \overline{d}, g\}$, $\mathcal{C}(\overline{b}) = \{f \vee g\}$, $\mathcal{C}(\overline{d}) = \{e, a \vee f\}$ and $\mathcal{C}(e) = \{\overline{d}\}$. With regard to $\mathcal{C}(a)$, we have $p_1(a) = 1$ because of the unary clause g, and $p_2(a) = 1$ owing to $f \vee \overline{d}$. Proceeding thus for all variables in the clauses of $\mathcal{I}(x)$, the value of $h(x)$ is :
$(2p_1(a)+p_2(a))(2p_1(\overline{b})+p_2(\overline{b}))+(2p_1(\overline{d})+p_2(\overline{d}))(2p_1(e)+p_2(e)) = (2+1)(1)+(2+1)(2) = 9$.
We assessed the performance of the heuristc $h(t)$ in a pure DPL procedure. At each node of the tree developed by the procedure, the product $h(x)h(\overline{x})$ was computed for every as yet unassigned variable x. The chosen branching variable was that corresponding to the maximum value of $h(x)h(\overline{x})$ at the node under consideration. The product $h(x)h(\overline{x})$ favors opposite literals which may each belong to a different backbone relative to a separate subset of clauses. The performance of this heuristic was compared to that of a heuristic of the type used in recent algorithms. We implemented the heuristic of the solver "*satz214*" [Li, 1999], appearing currently to be the fastest for solving random 3-SAT formulae. We solved samples of 200 random 3-SAT formulae having from 200 to 300 variables in steps of 25, and a clauses-to-variables ratio of 4.25. For every considered number of variables, the mean ratio of the computation time with the *satz214* heuristic compared to the computation time with the heuristic described above ("*h*" heuristic), for a level limited to 3, has been plotted on Figure 2. The smooth line connecting the consecutive points shows the evolution of this mean ratio from 200 to 300 variables. This ratio is seen to be markedly above 1, it increases from 1.2 to 1.3 with accelerated growth. Table 1

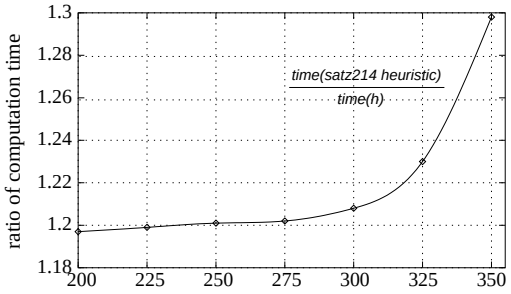

Figure 2: mean time ratio of *satz214* heuristic to "*h*" heuristic on hard random 3-SAT unsatisfiable formulae.

gives the mean size of the tree developed with either heuristic. These results immediately demonstrate the advantage of the approach based on the concept of backbone.

size of formulae in #vars	"*h*" heuristic mean #nodes	*satz214* heuristic mean #nodes
200	1556	1647
225	3165	3363
250	8229	8774
275	18611	20137
300	36994	41259
325	77744	93279
350	178813	252397

Table 1: average size of tree built by *satz214* heuristic and "*h*" heuristic on hard random 3-SAT unsatisfiable formulae.

2.2 Picking backbone variables

The technique described in this section aims at improving the efficiency of the previous heuristic by detecting either *literals* which *must* belong to some backbone relative to some set of clauses, or *clauses* that *cannot* be associated to literals belonging to a backbone. Only the principles of the technique are given in this section, together with illustrative examples.

A literal t has to belong to a backbone relative to some set of clauses, if among all possibilities leading to t being TRUE, at least one will necessarily hold. Take the simple example of the following clauses:

Example 1:

$(a \vee \overline{b}) \wedge (\overline{a} \vee h) \wedge (x \vee \overline{g} \vee \overline{h}) \wedge$
$(a \vee \overline{c}) \wedge (\overline{a} \vee g) \wedge (\overline{x} \vee e \vee f) \wedge (x \vee b \vee c)$

Consider the literal x. At the first level, the possibilities that it be true are $(\overline{g} \vee \overline{h} \equiv FALSE)$ and $(b \vee c \equiv FALSE)$. At the second level, these possibilities become simply $a \equiv FALSE$ and $\overline{a} \equiv FALSE$. Necessarily one or the other will obtain. Therefore, x belongs to the backbone associated to the given set of clauses.

Conversely, a literal t in some clause cannot belong to a backbone, if setting to TRUE, it leads to an inconsistency. This principle generalizes the classical local treatment of inconsistency detection at each node of a solving tree, such as implemented in *csat*, *posit*, and *satz214*. As an example, take the following clauses:

Example 2:

$(x \vee \beta) \wedge (\overline{x} \vee \overline{a} \vee b) \wedge (\overline{x} \vee c) \wedge$
$(\overline{x} \vee e) \wedge (x \vee \overline{\beta} \vee a) \wedge (\overline{e} \vee a) \wedge (\overline{c} \vee \overline{a} \vee \overline{b})$

According to the criteria usually adopted by algorithms in the literature, only the literal $\overline{x}$ or at most the 2 literals x and $\overline{x}$ in the clauses of Example 2 will be selected for a local treatment of inconsistency detection. Setting x to TRUE leads to an inconsistency by unit propagation. From the backbone viewpoint, x cannot, therefore, belong to a backbone. As mentioned, this case is classically handled by existing algorithms. Now suppose that, instead of the sixth clause $(\overline{e} \vee a)$ we have the clause $(\overline{e} \vee a \vee \beta)$. Unit propagation no longer detects an inconsistency. On the other hand, following our backbone viewpoint and without changing the classical selection criteria for the local treatment of inconsistencies (which are of recognized efficiency), it is possible to detect that the first clause containing x, $(x \vee \beta)$, cannot contribute to a set of clauses such that x would belong to a backbone relative to this set. Indeed, the only possibility that x be TRUE is that β be FALSE. An inconsistency now follows, again by unit propagation. As a consequence, β is set to TRUE, and the clause $(x \vee \beta)$ cannot be associated to a backbone including x. The technique inspired by the principle just stated is called *pickback*.

We briefly mention two other examples which show the general nature of this principle.

Example 3:

$(\overline{x} \vee a) \wedge (x \vee \beta \vee \gamma) \wedge (\overline{b} \vee d \vee \gamma) \wedge (\overline{c} \vee \overline{e}) \wedge$
$(\overline{x} \vee b) \wedge (\overline{a} \vee \beta \vee c) \wedge (\overline{c} \vee e \vee f) \wedge (\overline{d} \vee \gamma \vee \overline{f})$

Setting x to TRUE does not lead to an inconsistency. Yet, setting $(\beta \vee \gamma)$ to FALSE by 'pickback' leads, via unit propagation, to an inconsistency. We may therefore add this clause, $(\beta \vee \gamma)$, which subsumes $(x \vee \beta \vee \gamma)$.

Example 4:

$(\overline{x} \vee b) \wedge (x \vee \beta \vee \gamma) \wedge (\gamma \vee f) \wedge (\overline{\gamma} \vee e) \wedge$
$(\overline{x} \vee a) \wedge (\overline{a} \vee d \vee c) \wedge (\overline{e} \vee \beta) \wedge (\overline{f} \vee \overline{c}) \wedge (\overline{b} \vee c \vee \overline{d})$

Setting x to TRUE does not lead to an inconsistency. Yet setting β alone to FALSE in $(\beta \vee \gamma)$ allows to deduct the value FALSE for γ, thus validating a stronger pickback than in the previous example.

3 Experimental results

The "*h*" heuristic and the technique just described in section 2 were implemented in a DPL procedure. This implementation is denoted *cnfs* (short for CNF solver). In this section comparative performance results are given, pitting *cnfs* against 4 solvers which currently appear to be the best performers on k-SAT or 3-SAT random formulae. These are: TABLEAU in its version *ntab* [Crawford and Auton, 1993], *posit* [Freeman, 1996], *csat* [Dubois *et al.*, 1996], and *satz214*[1] [Li, 1999]. In addition we consider 2 strong solvers *sato* [Zhang, 1997], *rel_sat* [Bayardo and Schrag, 1996] which are more devoted to structured formulae than to random formulae. We carried out our experimentations on random 3-SAT formulae generated as per the classical generators in the literature (i.e. each clause drawn independently and built by randomly drawing 3 among a fixed number n of variables, then negating each with probability $\frac{1}{2}$). In order that the impact of the backbone approach used in *cnfs* be shown with as much precision as possible, our experiments covered a wide range of formula sizes, going from 200 to 700 variables and drawing large samples. The hardware consisted of AMD Athlon-equipped[2] PCs running at 1 GHz under a Linux operating system. The comparative results thus obtained are now to be given in detail.

3.1 Performance comparison results

Comparative results from 200 to 400 variables

Table 2 gives the mean sizes of the solving trees developed by the 7 programs under scrutiny, *sato, rel_sat, ntab, csat, posit, satz214* and *cnfs*, on samples of 1000 formulae with 200 and 300 variables, and on samples of 500 formulae with 400 variables. Mean tree sizes are expressed as numbers of branching nodes, and are listed separately for satisfiable and unsatisfiable formulae. They are found to be from 8 to about 1.5 times smaller for *cnfs* than for the 4 solvers *ntab, csat, posit* and *satz214* in the case of unsatisfiable formulae, and from 9 to 1.6 times smaller for satisfiable formulae; these ratios depend on both solver and formula size. It is important to observe that the compute time ratio of *cnfs* with respect to each of the other 6 solvers increases with formula size. Table 3 shows how improved tree sizes obtained with *cnfs* translate into computation times. Compute time ratios of *cnfs* with respect to *ntab, csat, posit* and *satz214* are seen to vary from 26 to 1.3 in the unsatisfiable, and from 29 to 1.75 in the satisfiable case. Tree size improvements thus carry over into equivalent computation time gains. These first results show, therefore, that *cnfs* provides, in the range of formulae from

[1] The new *satz215* has, from our first experiments, similar performances on Random 3-SAT formulae as *satz214*

[2] Advanced Micro Devices Inc. (http://www.amd.com)

SAT Solvers	unsat (N) & sat (Y)	200 V 850 C (482 N) (518 Y) mean #nodes (*std dev.*)	300 V 1275 C (456 N) (534 Y) mean #nodes (*std dev.*)	400 V 1700 C (245 N) (255 Y) mean #nodes in millions (*std dev.*)
sato	unsat	14076 (*5225*)	443313 (*109694*)	-
	sat	4229 (*4754*)	151728 (*157102*)	-
rel_sat	unsat	1319 (*535*)	56064 (*26136*)	2.2 (*0.9*)
	sat	478 (*452*)	18009 (*19217*)	0.82 (*0.88*)
ntab	unsat	973 (*764*)	78042 (*33222*)	3.0 (*1.4*)
	sat	756 (*673*)	27117 (*27790*)	1.1 (*1.2*)
csat	unsat	2553 (*997*)	90616 (*37729*)	3.6 (*1.7*)
	sat	733 (*763*)	26439 (*31224*)	1.8 (*2.4*)
posit	unsat	1992 (*754*)	82572 (*35364*)	3.4 (*1.7*)
	sat	789 (*694*)	34016 (*31669*)	1.5 (*1.4*)
satz214	unsat	623 (*206*)	18480 (*7050*)	0.56 (*0.23*)
	sat	237 (*216*)	6304 (*6541*)	0.21 (*0.21*)
cnfs	unsat	470 (*156*)	12739 (*4753*)	0.37 (*0.15*)
	sat	149 (*154*)	3607 (*4089*)	0.12 (*0.13*)

Table 2: average size of tree on hard random 3-SAT formulae from 200 to 400 variables for *sato*, *rel_sat*, *TABLEAU*, *csat*, *POSIT*, *satz214* & *cnfs* solvers

SAT Solvers	unsat (N) & sat (Y)	200 V 850 C (482 N) (518 Y) mean time (*std dev.*)	300 V 1275 C (456 N) (534 Y) mean time (*std dev.*)	400 V 1700 C (245 N) (255 Y) mean time (*std dev.*)
sato	unsat	89.2s (*84.6s*)	18h 27m (*7h*)	-
	sat	13.2s (*38s*)	4h (*6h 40m*)	-
rel_sat	unsat	4.5s (*1.9s*)	343s (*169s*)	6h 29m (*2h 47m*)
	sat	1.6s (*1.6s*)	114.3s (*123s*)	2h 30m (*2h 40m*)
ntab	unsat	0.81s (*0.30s*)	48.0s (*20.6s*)	48m 15s (*23m 21s*)
	sat	0.38s (*0.28s*)	16.6s (*16.8s*)	18m 5s (*19m 16s*)
csat	unsat	0.47s (*0.20s*)	29.6s (*13.3s*)	29m 1s (*14m 36s*)
	sat	0.15s (*0.15s*)	9.0s (*10.7s*)	7m 47s (*10m 1s*)
posit	unsat	0.28s (*0.1s*)	15.1s (*6.6s*)	13m 7s (*6m 35s*)
	sat	0.11s (*0.1s*)	6.3s (*5.9s*)	5m 56s (*5m 36s*)
satz214	unsat	0.17s (*0.05s*)	4.9s (*1.8s*)	2m 44s (*1m 09s*)
	sat	0.07s (*0.05s*)	1.7s (*1.7s*)	1m 05s (*1m 03s*)
cnfs	unsat	0.13s (*0.04s*)	3.7s (*1.3s*)	1m 50s (*0m 43s*)
	sat	0.04s (*0.04s*)	1.1s (*1.2s*)	0m 37s (*0m 40s*)

Table 3: mean computation time on hard random 3-SAT formulae from 200 to 400 variables for *sato*, *rel_sat*, *TABLEAU*, *csat*, *POSIT*, *satz214* & *cnfs* solvers

200 to 400 variables, significant performance improvements that increase with formula size, in terms of tree size as well as computation time.

Further performance comparisons with *satz214*

Pursuing this line of experimentation, we now offer performance comparison results on formulae from 400 to 600 variables with the *satz214* solver, which performs best of the 6 against which *cnfs* was measured up to now.

We first solved samples of 100 formulae from 400 to 600 variables in increments of 20.

Figure 3 shows the mean computation time curves (plotted in the same way as for the Figure 2) of *satz214* and *cnfs* for the unsatisfiable formulae within the solved samples. Indeed, the most significant performance figure for a complete algorithm is on unsatisfiable formulae. The gain previously observed between 200 and 400 variables is seen further to increase from 400 to 600. *cnfs* is 1.85 times faster than *satz214* for 500 variables, and 2.78 times faster for 600.

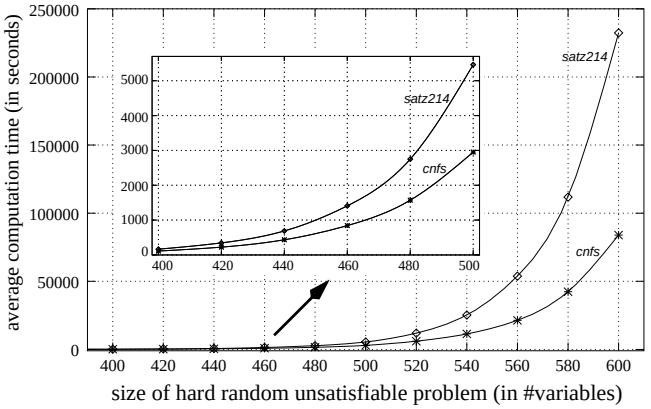

Figure 3: mean computation time of *satz214* & *cnfs* on 3-SAT hard random unsatisfiable formulae from 400 to 600 variables

Table 4 contains precise compared mean values of tree size and computation time on formulae with 500 and 600 variables, listing the satisfiable and unsatisfiable cases separately. Tree size gain increase of *cnfs* versus *satz214* may be noticed not to carry over entirely into computation time gain increase. Indeed, the gain factor for *cnfs* on unsatisfiable formulae goes from 2 for 500 variables to 4.8 for 600. This probably reflects the high cost of the backbone-variable search heuristic. One can therefore hope that technical improvements of the latter will lead to yet greater computation time performance gains. Finally, Figure 4 sums up, on the whole range 200 to 600 variables, the evolution of the gain ratio of *cnfs* vs *satz214* in computation time and tree size. These curves clearly demonstrate a complexity function gain of *cnfs* over *sat214* on random 3-SAT formulae.

#Vars #Clauses	unsat (N) sat (Y)	mean #nodes in millions (*std dev.*)		mean time (*std dev.*)	
		cnfs	*satz214*	*cnfs*	*satz214*
500 V	58 N	8 (*3.3*)	16.6 (*7.2*)	49m 10s (*19s*)	91m 02s (*39s*)
2125 C	62 Y	2.5 (*3.8*)	6.5 (*7.2*)	15m 24s (*23s*)	36m 21s (*40s*)
600 V	48 N	178 (*71*)	857 (*373*)	23h 18m (*9h*)	64h 32m (*28h*)
2550 C	52 Y	37.8 (*50*)	233 (*315*)	4h 54m (*6.5h*)	18h 20m (*24h*)

Table 4: average size of tree and mean computation time on hard random 3-SAT formulae from 500 and 600 variables

Solving hard 3-SAT formulae up to 700 variables

For the reader's information, Table 5 shows how *cnfs* performs in computation time and number of nodes on two samples of 30 formulae with 650 and 700 variables, respectively. Formulae with 700 variables, regarded as quite large for complete solving methods [Selman *et al.*, 1997] are now within reach, and this in approximately 600 machine hours on a single-processor 'domestic' PC. Let us also indicate that for random formulae with 700 variables, irrespective of their satisfiability, the mean solving time with *cnfs* is about 300 hours.

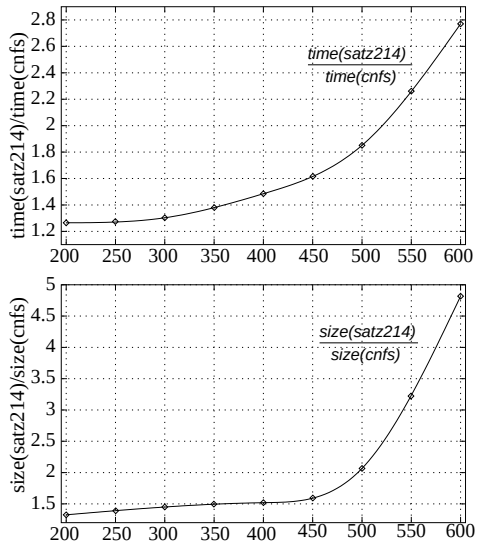

Figure 4: progression of $\frac{time(satz214)}{time(cnfs)}$ and $\frac{size(satz214)}{size(cnfs)}$ on unsatisfiable formulae from 200 to 600 variables

#Vars #Clauses	unsat (N) sat (Y)	mean #nodes in millions (*std dev.*)	mean time (*std dev.*)
650 V	13 N	1140 (*587*)	5 days 21h (*71h*)
2762 C	17 Y	241 (*399*)	1 day 6h (*50h*)
700 V	12 N	4573 (*1703*)	26 days 4h (*228h*)
2975 C	18 Y	653 (*922*)	3 days 19h (*129h*)

Table 5: mean computation time and average size of tree of *cnfs* on large hard 3-SAT formulaes up to 700 variables

4 Conclusion

In the course of the last decade, algorithms based on the DPL procedure for solving propositional formulae have seen a performance improvement that was quite real, if much less spectacular than in the case of stochastic algorithms. It was recently suggested that the performance of DPL-type algorithms might be close to their limitations, giving rise to a fear that subsequent progress might be very difficult to achieve and that large unsatisfiable formulae (e.g., 700 variables) might remain beyond reach. We have presented a DPL-type algorithm incorporating mainly a new and simple heuristic using the backbone concept recently introduced from models of physics. This concept has changed the viewpoint from which classical heuristics were developed. We were thus able to improve the current best solving performance for hard 3-SAT formulae by a ratio increasing with formula size (equal to 3 for 600 variables), and we have shown that solving unsatisfiable formulae with 700 variables was feasible. An important lesson can be drawn from our results. In order to improve the performance of DPL-type algorithms significantly and to enhance the state of the art in complete solving, it appears that a deep understanding of the structure of the solutions of a SAT formula is paramount. This is why experimental, as well as theoretical studies aiming to further such comprehension are essential.

References

[Bayardo and Schrag, 1996] R. Bayardo and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. of the 14th Nat. Conf. on Artificial Intelligence*, pages 203–208. AAAI, 1996.

[Bollobás *et al.*, 2000] B. Bollobás, C. Borgs, J. T. Chayes, J. H. Kim, and D. B. Wilson. The scaling window of the 2-SAT transition. *Random Structures and Algorithms*, 2000.

[Boufkhad and Dubois, 1999] Y. Boufkhad and O. Dubois. Length of prime implicants and number of solutions of random CNF formulae. *Theoretical Computer Science*, 215 (1–2):1–30, 1999.

[Crawford and Auton, 1993] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proc. of the 11th Nat. Conf. on Artificial Intelligence*, pages 21–27. AAAI, 1993.

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. Assoc. for Comput. Mach.*, (5):394–397, 1962.

[Dubois *et al.*, 1996] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. *DIMACS Series in Discrete Math. and Theoret. Comp. Sc.*, pages 415–436, 1996.

[Freeman, 1996] J. W. Freeman. Hard random 3-SAT problems and the Davis-Putnam procedure. *Artificial Intelligence*, 81(1–2):183–198, 1996.

[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 & Company, San Francisco, 1979.

[Gent and Walsh, 1994] I. P. Gent and T. Walsh. The SAT phase transition. In *Proc. of the 11th European Conf. on Artificial Intelligence*, pages 105–109. ECAI, 1994.

[Li and Gerard, 2000] C. M. Li and S. Gerard. On the limit of branching rules for hard random unsatisfiable 3-SAT. In *Proc. of European Conf. on Artificial Intelligence*, pages 98–102. ECAI, 2000.

[Li, 1999] C. M. Li. A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, 71(2):75–80, 1999.

[Mitchell *et al.*, 1992] D. Mitchell, B. Selman, and H. J. Levesque. Hard and easy distribution of SAT problems. In *Proc. 10th Nat. Conf. on Artificial Intelligence*. AAAI, 1992.

[Monasson *et al.*, 1999] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature 400*, pages 133–137, 1999.

[Selman *et al.*, 1997] B. Selman, H. Kautz, and D. McAllester. Ten challenges in propositional reasoning and search. In *Proc. of IJCAI-97*, 1997.

[Zhang, 1997] H. Zhang. SATO. an efficient propositional prover. In *Proc. of the 14th Internat. Conf. on Automated Deduction*, pages 272–275. CADE-97, LNCS, 1997.

Backbones in Optimization and Approximation

John Slaney
Automated Reasoning Project
Australian National University
Canberra
Australia
jks@arp.anu.edu.au

Toby Walsh
Department of Computer Science
University of York
York
England
tw@cs.york.ac.uk

Abstract

We study the impact of backbones in optimization and approximation problems. We show that some optimization problems like graph coloring resemble decision problems, with problem hardness positively correlated with backbone size. For other optimization problems like blocks world planning and traveling salesperson problems, problem hardness is weakly and negatively correlated with backbone size, while the cost of finding optimal and approximate solutions is positively correlated with backbone size. A third class of optimization problems like number partitioning have regions of both types of behavior. We find that to observe the impact of backbone size on problem hardness, it is necessary to eliminate some symmetries, perform trivial reductions and factor out the effective problem size.

1 Introduction

What makes a problem hard? Recent research has correlated problem hardness with rapid transitions in the solubility of decision problems [Cheeseman *et al.*, 1991; Mitchell *et al.*, 1992]. The picture is, however, much less clear for optimization and approximation problems. Computational complexity provides a wealth of (largely negative) worst-case results for decision, optimization and approximation. Empirical studies like those carried out here add important detail to such theory. One interesting notion, borrowed from statistical physics, is that of the *backbone*. A percolation lattice, which can be used as a model of fluid flow or forest fires, undergoes a rapid transition in the cluster size at a critical threshold in connectivity. The backbone of such a lattice consists of those lattice points that will transport fluid from one side to the other if a pressure gradient applied. The backbone is therefore the whole cluster minus any dead ends. The size and structure of this backbone has a significant impact on the properties of the lattice. In decision problems like propositional satisfiability, an analogous notion of "backbone" variables has been introduced and shown to influence problem hardness [Monasson *et al.*, 1998]. Here, we extend this notion to optimization and approximation and study its impact on the cost of finding optimal and approximate solutions.

2 Backbones

In the satisfiability (SAT) decision problem, the *backbone* of a formula φ is the set of literals which are true in every model [Monasson *et al.*, 1998]. The size of the backbone and its fragility to change have been correlated with the hardness of SAT decision problems [Parkes, 1997; Monasson *et al.*, 1998; Singer *et al.*, 2000a; Achlioptas *et al.*, 2000]. A variable in the backbone is one to which it is possible to assign a value which is absolutely *wrong* – such that no solution can result no matter what is done with the other variables. A large backbone therefore means many opportunities to make mistakes and to waste time searching empty subspaces before correcting the bad assignments. Put another way, problems with large backbones have solutions which are clustered, making them hard to find both for local search methods like GSAT and WalkSAT and for systematic ones like Davis-Putnam.

The notion of backbone has been generalized to the decision problem of coloring a graph with a fixed number of colors, k [Culberson and Gent, 2000]. As we can always permute the colors, a pair of nodes in a k-colorable graph is defined to be *frozen* iff each has the same color in every possible k-coloring. No edge can occur between a frozen pair. The *backbone* is then simply the set of frozen pairs.

To generalize the idea of a backbone to optimization problems, we consider a general framework of assigning values to variables. The backbone is defined to be the set of *frozen decisions*: those with fixed outcomes for all optimal solutions. In some cases, "decision" just amounts to "assignment": for example, in MAX-SAT, the backbone is simply the set of assignments of values to variables which are the same in every possible optimal solution. In general, however, the relevant notion of decision is obtained by abstraction over isomorphism classes of assignments. In graph coloring, for example, the decision to color two nodes the same is a candidate for being in the backbone whereas the actual assignment of "blue" to them is not because a trivial permutation of colors could assign "red" instead.

3 Graph coloring

We first consider the optimization problem of finding the minimal number of colors needed to color a graph. A pair of nodes in a graph coloring optimization problem is *frozen* iff each has the same color in every possible optimal coloring. No edge can occur between a frozen pair without increasing the chromatic number of the graph. The *backbone* is again

the set of frozen pairs. In a graph of n nodes and e edges, we normalize the size of the backbone by $n(n-1)/2 - e$, the maximum possible backbone size. As with graph coloring decision problems [Culberson and Gent, 2000], we investigate the "frozen development" by taking a random list of edges and adding them to the graph one by one, measuring the backbone size of the resulting graph. We study the frozen development in single instances since, as with graph coloring decision problems [Culberson and Gent, 2000], averaging out over an ensemble of graphs obscures the very rapid changes in backbone size.

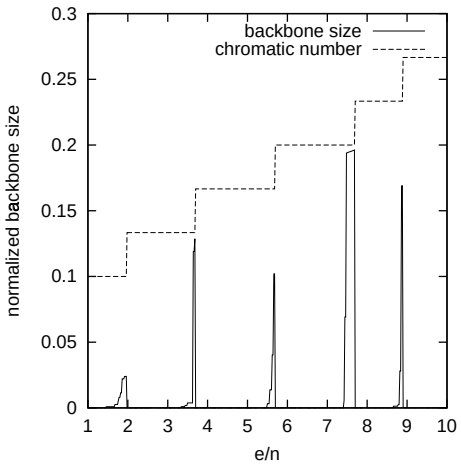

Figure 1: Frozen development in a single 50 node graph. Backbone size (y-axis) is plotted against e/n. The number of edges e is varied from n to $10n$ in steps of 1. Backbone size is normalized by its maximum value. The chromatic number, which increases from 2 to 7, is plotted on the same axes.

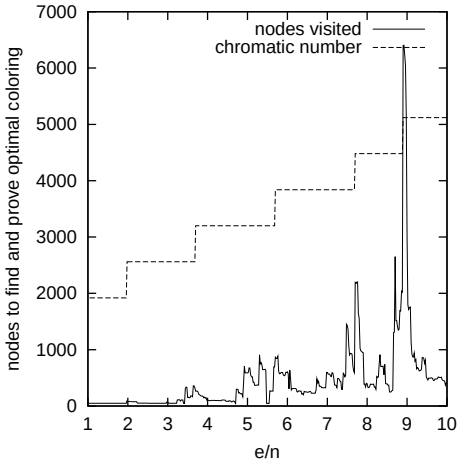

Figure 2: Cost to color graph optimally for the same graphs as Figure 1. Nodes visited (y-axis) is plotted against e/n.

In Figure 1, we plot the frozen development for a typical 50 node graph. Just before the chromatic number increases, there are distinct peaks in backbone size. When the chromatic number increases, the backbone size immediately collapses. In Figure 2, we plot the search cost to find the optimal coloring for the same 50 node graph. To find optimal colorings, we use an algorithm due to Mike Trick which is based upon Brelaz's DSATUR algorithm [Brelaz, 1979]. Search cost peaks with the increases in chromatic number and the peaks in the backbone size. Optimization here closely resembles decision since it is usually not hard to prove that a coloring is optimal. There is thus a strong correlation between backbone size and both optimization and decision cost.

4 Traveling salesperson problem

We next consider the traveling salesperson problem. A leg in a traveling salesperson (TSP) optimization problem is *frozen* iff it occurs in every possible optimal tour. The TSP *backbone* is simply the set of frozen legs. We say that the TSP backbone is *complete* iff it is of size n. In such a situation, the optimal tour is unique. Note that it is impossible to have a backbone of size $n-1$. It is, however, possible to have a backbone of any size $n-2$ or less. Computing the TSP backbone highlights a connection with sensitivity analysis. A leg occurs in the backbone iff adding some distance, ϵ to the corresponding entry in the inter-city distance matrix increases the length of the optimal tour. A TSP problem with a large backbone is therefore more sensitive to the values in its inter-city distance matrix than a TSP problem with a small backbone.

To explore the development of the backbone in TSP optimization problems, we generated 2-D integer Euclidean problems with 20 cities randomly placed in a square of length l. We varied $\log_2(l)$ from 2 to 18, generating 100 problems at each integer value of $\log_2(l)$, and found the backbone and the optimal tour using a branch and bound algorithm based on the Hungarian heuristic. The cost of computing the backbone limited the experiment to $n = 20$. The backbone quickly becomes complete as $\log_2(l)$ is increased. Figure 3 is a scatter plot of backbone size against the cost to find and prove the tour optimal.

The Pearson correlation coefficient, r between the normalized backbone size and the log of the number of nodes visited to find and prove the tour optimal is just -0.0615. This suggests that there is a slight negative correlation between backbone size and TSP optimization cost. We took the log of the number of nodes visited as it varies over 4 orders of magnitude. This conclusion is supported by the Spearman rank correlation coefficient, ρ which is a distribution free test for determining whether there is a monotonic relation between two variables. The data has a Spearman rank correlation of just -0.0147.

To explore the difference between optimization and decision cost, in Figure 4 we plot the (decision) cost for finding the optimal tour. The Pearson correlation coefficient, r between the normalized backbone size and the log of the number of nodes visited to find the optimal tour is 0.138. This suggests that there is a positive correlation between backbone size and TSP decision cost. This conclusion is supported by the Spearman rank correlation coefficient, ρ which is 0.126.

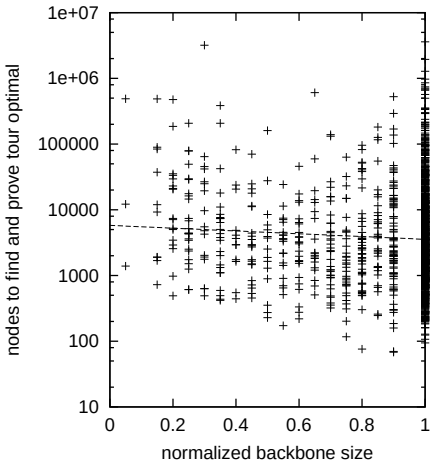

Figure 3: Cost to find and prove the tour optimal (y-axis, logscale) against normalized backbone size (x-axis) for 2-D integer Euclidean TSP problems with 20 cities placed on a square of length l. Nodes visited by a branch and bound algorithm (y-axis, logscale) is plotted against normalized backbone size (x-axis). 100 random problems are generated at each integer value of $\log_2(l)$ from 2 to 18. The straight line gives the least squares fit to the data.

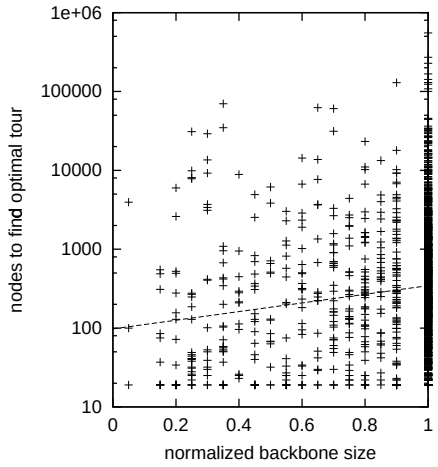

Figure 4: Cost to find optimal tour (y-axis, logscale) against normalized backbone size (x-axis) for the 20 city problems from Figure 3. The straight line again gives the least squares fit to the data.

TSP is unlike graph coloring in that optimization appears significantly different from decision. We conjecture this is a result of there usually being no easy proofs of tour optimality. Indeed, the cost of proving tours optimal is negatively correlated with backbone size. This roughly cancels out the positive correlation between the (decision) cost of finding the optimal tour and backbone size. But why does the cost of proving tours optimal negatively correlated with the backbone size? If we have a small backbone, then there are many optimal and near-optimal tours. An algorithm like branch and bound will therefore have to explore many parts of the search space before we are sure that none of the tours is any smaller.

5 Number partitioning

We have seen that whether optimization problems resemble decision problems appears to depend on whether there are cheap proofs of optimality. Number partitioning provides a domain with regions where proofs of optimality are cheap (and there is a positive correlation between optimization cost and backbone size), and regions where proofs of optimality are typically not cheap (and there is a weak negative correlation between optimization cost and backbone size).

One difficulty in defining the backbone of a number partitioning problem is that different partitioning algorithms make different types of decisions. For example, Korf's CKK algorithm decides whether a pair of numbers go in the same bin as each other or in opposite bins [Korf, 1995]. One definition of backbone is thus those pairs of numbers that cannot be placed in the same bin or that cannot be placed in opposite bins. By comparison, the traditional greedy algorithm for number partitioning decides into which bin to place each number. Another definition of backbone is thus those numbers that must be placed in a particular bin. We can break a symmetry by irrevocably placing the largest number in the first bin. Fortunately, the choice of definition does not appear to be critical as we observe very similar behavior in normalized backbone size using either definition. In what follows, we therefore use just the second definition.

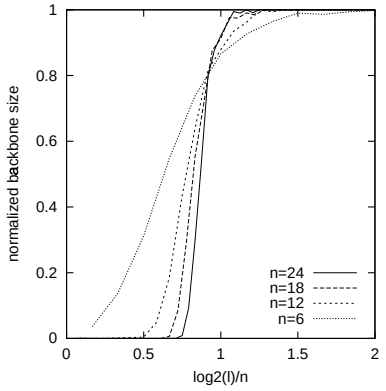

Figure 5: Frozen development averaged over 100 problems. Average backbone size (y-axis) against $\log_2(l)/n$ (x-axis). Problems contain n random numbers in the interval $[0, l)$, and $\log_2(l)$ is varied from 0 to $2n$ in steps of 1. Backbone size is normalized by its maximum value.

In Figure 5, we plot the frozen development averaged over 100 problems. The frozen development in a single problem is very similar. As in [Gent and Walsh, 1998], we partition n random numbers uniformly distributed in $[0, l)$. We generate 100 problem instances at each n and $l_{\max}$, and then prune numbers to the first $\log_2(l)$ bits using mod arithmetic. The size of the optimal partition is therefore monotonically

increasing with l. We see characteristic phase transition behaviour in the backbone size. There is a very sharp increase in backbone size in the region $0.6 < \log_2(l)/n < 1$ where even the best heuristics like KK fail to find backtrack free solutions. By the decision phase boundary at $\log_2(l)/n \approx 0.98$ [Gent and Walsh, 1998], the backbone tends to be complete and the optimal solution is therefore unique. This rapid transition in average backbone size should be compared to graph coloring where [Culberson and Gent, 2000] typically had to look at single instances to see large jumps in backbone size.

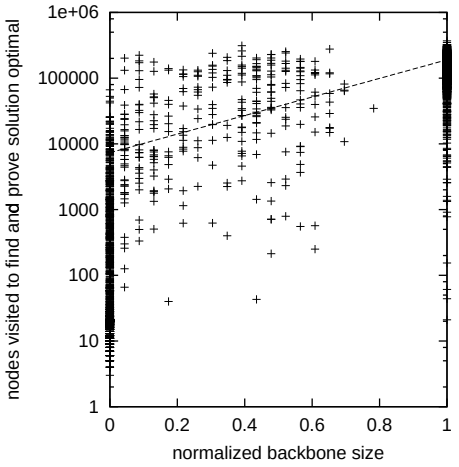

Figure 6: Optimization cost (y-axis, logscale) against normalized backbone size (x-axis) for the $n = 24$ number partitioning problems from Figure 5. The straight line gives the least squares fit to those problems whose backbone is neither complete nor empty.

In Figure 6, we give a scatter plot for the optimization cost for Korf's CKK algorithm against backbone size. The data falls into two regions. In the first, optimization problems have backbones less than 80% complete. Optimization in this region is similar to decision as proofs of optimality are typically easy, and optimization cost is positively correlated with backbone size. Data from this region with non-empty backbones has a Pearson correlation coefficient, r of 0.356, and a Spearman rank correlation coefficient, ρ of 0.388. In the second region, optimization problems have complete backbones. The cost of proving optimality is now typically greater than the cost of finding the optimal solution. Due to the rapid transition in backbone size witnessed in Figure 5, we observed no problems with backbones between 80% and 100% complete.

6 Blocks world planning

Our fourth example taken from the field of planning raises interesting issues about the definition of a backbone. It also highlights the importance of considering the "effective" problem size and of eliminating trivial aspects of a problem.

We might consider a solution to a blocks world planning problem to be the plan and the backbone to be those moves present in all optimal (minimal length) plans. However, since most moves simply put blocks into their goal positions and are therefore trivially present in all plans, almost all of a plan is backbone. A more informative definition results from considering "deadlocks". A deadlock is a cycle of blocks, each of which has to be moved before its successor can be put into the goal position. Each deadlock has to be broken, usually by putting one of the blocks on the table. Once the set of deadlock-breaking moves has been decided, generating the plan is an easy (linear time) problem [Slaney and Thiébaux, 1996]. A better definition of solution then is the set of deadlock-breaking moves. However, this is not ideal as many deadlocks contain only one block. These singleton deadlocks give forced moves which inflate the backbone, yet are detectable in low-order polynomial time and can quickly be removed from consideration. We therefore define a solution as the set of deadlock-breaking moves in a plan, excluding those which break singleton deadlocks. The backbone is the set of such moves that are in every optimal solution.

We considered uniformly distributed random blocks world problems of 100 blocks, with both initial and goal states completely specified. To obtain optimal solutions, we used the domain-specific solver reported in [Slaney and Thiébaux, 1996] and measured hardness as the number of branches in the search tree. As in [Slaney and Thiébaux, 1998], we observed a cost peak as the number of towers in the initial and goal states reaches a critical value of 13–14 towers. We therefore plotted backbone size against the number of towers, and found that this peaks around the same point (see Figure 7).

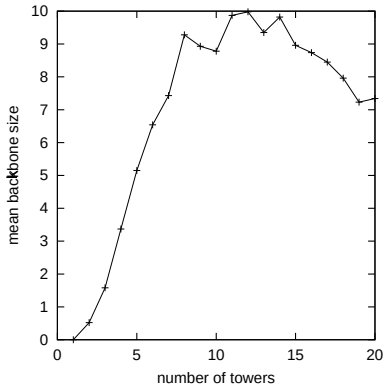

Figure 7: Mean backbone size for 100 block optimization problems against number of towers in initial and goal state.

Although problem hardness is in a sense correlated with backbone size, this result must be interpreted carefully because solution size also peaks at the same point. With more than about 13–14 towers, few deadlocks exist so solution size, as we measure it, is small. With a smaller number of towers, the problem instance is dominated by singleton deadlocks, so again the solution is small. The size of the backbone as a proportion of the solution size shows almost no dependence on the number of towers.

Another important feature of the blocks world is that the number of blocks in an instance is only a crude measure of problem "size". At the heart of a blocks world planning problem is the sub-problem of generating a hitting set for a

collection of deadlocks. The *effective* size of an instance is therefore the number of blocks that have to be considered for inclusion in this hitting set. This effective size dominates the solution cost, overshadowing any effect of backbone size. In our next experiment, therefore, we filtered out all but those instances of effective size 35. We obtain similar results restricting to other sizes. Of 9000 random problems, 335 were of effective size 35. For each of those, we measured the hardness of solving two decision problems: whether there exists a plan of length l_{opt} (the optimal plan length), and whether there exists a plan of length $l_{opt} - 1$. These can be regarded as measuring the cost of finding an optimal solution and of proving optimality respectively.

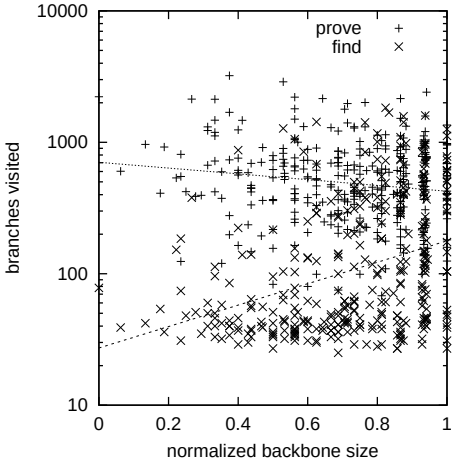

Figure 8: Cost of finding an optimal solution ($\times$) and of proving optimality ($+$) against backbone size as a proportion of solution size, for 100 block problems of effective size 35. The straight lines give the least squares fits to the data.

Figure 8 shows the results are similar to TSP problems. Finding an optimal solution tends to be harder if the backbone is larger, for the familiar reason that if solutions are clustered, most of the search space is empty. This data has a Pearson correlation coefficient, r of 0.357 and a Spearman rank correlation coefficient, ρ of 0.389. Proving optimality, on the other hand, tends to be slightly easier with a larger backbone. This data has $r = -0.128$ and $\rho = -0.086$.

7 Approximation and ϵ-backbones

Our definition of backbone ignores those solutions which are close to optimal. In many real-world situations, we are willing to accept an approximate solution that is close to optimal. We therefore introduce the notion of the ϵ-*backbone*: the set of frozen decisions in all solutions within a factor $(1 - \epsilon)$ of optimal. For $\epsilon = 0$, this gives the previous definition of backbone. For $\epsilon = 1$, the ϵ-backbone is by definition empty. For example, the TSP ϵ-backbone consists of those legs which occur in all tours of length less than or equal to $l_{opt}/(1 - \epsilon)$.

In Figure 9, we give a scatter plot of the size of the $1/2$-backbone for number partitioning problems against the cost of finding an approximate solution within a factor 2 of optimal. Similar plots are seen for other values of ϵ. As with $\epsilon = 0$, the data falls into two regions. In the first, problems have 1/2-backbones less than 80% complete. The cost of approximation in this region is positively correlated with backbone size. However, the correlation is less strong than that between backbone size and optimization cost. Data from this region with non-empty backbones has a Pearson correlation coefficient, r of 0.152, and a Spearman rank correlation coefficient, ρ of 0.139. In the second region, problems have complete 1/2-backbones. The cost of finding an approximate solutions in this region is now typically as hard as that for the hardest problems with incomplete 1/2-backbones.

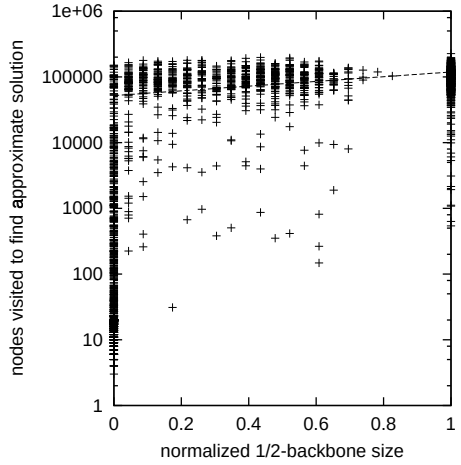

Figure 9: Cost of finding approximate solution within factor 2 of optimal (y-axis, logscale) against normalized $1/2$-backbone size (x-axis) for the $n = 24$ number partitioning problems from Figure 5. The straight line gives the least squares fit to those problems whose backbone is neither complete nor empty.

8 Related work

First moment methods can be used to show that, at the satisfiability phase transition, the expected number of solutions for a problem is exponentially large. Kamath *et al.* proved that, whilst most of these problems have few or no solutions, a few have a very large number of clustered solutions [Kamath *et al.*, 1995]. This was verified empirically by Parkes who showed that many variables are frozen although some are almost free [Parkes, 1997]. He argued that such problems are hard for local search algorithms to solve as solutions are clustered and not distributed uniformly over the search space.

Monasson *et al.* introduced the 2+p-SAT problem class to study computational complexity in NP-complete decision problems [Monasson *et al.*, 1998]. For $p < p_0 \approx 0.41$, random 2+p-SAT behaves like the polynomial random 2-SAT problem, whilst for $p > p_0$, random 2+p-SAT behaves like the NP-complete random 3-SAT problem [Monasson *et al.*, 1998; Singer *et al.*, 2000b]. The rapid change in backbone

size is continuous (second order) for $p < p_0$, and discontinuous (first order) for $p > p_0$. This transition may explain the onset of problem hardness and could be exploited in search.

Backbones have also been studied in TSP (approximation) problems [Kirkpatrick and Toulouse, 1985; Boese, 1995]. For example, Boese shows that optimal and near-optimal tours for the well known ATT 532-city problem tended are highly clustered [Boese, 1995]. Heuristic optimization methods for the TSP problem have been developed to identify and eliminate such backbones [Schneider *et al.*, 1996].

A related notion to the backbone in satisfiability is the spine [Bollobas *et al.*, 2001]. A literal is in the spine of a set of clauses iff there is a satisfiable subset in all of whose models the literal is false. For satisfiable problems, the definitions of backbone and spine coincide. Unlike the backbone, the spine is monotone as adding clauses only ever enlarges it.

9 Conclusions

We have studied backbones in optimization and approximation problems. We have shown that some optimization problems like graph coloring resemble decision problems, with problem hardness positively correlated with backbone size and proofs of optimality that are usually easy. With other optimization problems like blocks world and TSP problems, problem hardness is weakly and negatively correlated with backbone size, and proofs of optimality that are usually very hard. The cost of finding optimal and approximate solutions tends, however, to be positively correlated with backbone size. A third class of optimization problem like number partitioning have regions of both types of behavior.

What general lessons can be learnt from this study? First, backbones are often an important indicator of hardness in optimization and approximation as well as in decision problems. Second, (heuristic) methods for identifying backbone variables may reduce problem difficulty. Methods like randomization and rapid restarts [Gomes *et al.*, 1998] may also be effective on problems with large backbones. Third, it is essential to eliminate trivial aspects of a problem, like symmetries and decisions which are trivially forced, before considering its hardness. Finally, this and other studies have shown that there exist an number of useful parallels between computation and statistical physics. It may therefore pay to map over other ideas from areas like spin glasses and percolation.

Acknowledgements

The second author is an EPSRC advanced research fellow. We wish to thank the members of the APES research group.

References

[Achlioptas *et al.*, 2000] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proc. of 17th Nat. Conf. on AI*. 2000.

[Boese, 1995] K.D. Boese. Cost versus distance in the traveling salesman problem. Technical Report CSD-950018, UCLA Computer Science Department, 1995.

[Bollobas *et al.*, 2001] B. Bollobas, C. Borgs, J. Chayes, J.H. Kim, and D.B. Wilson. The scaling window of the 2-SAT transition. *Random Structures and Algorithms*, 2001. To appear. Available from `dimacs.rutgers.edu/~dbwilson/2sat/`.

[Brelaz, 1979] D. Brelaz. New methods to color the vertices of a graph. *Communications of ACM*, 22:251–256, 1979.

[Cheeseman *et al.*, 1991] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proc. of the 12th IJCAI*, pages 331–337. 1991.

[Culberson and Gent, 2000] J. Culberson and I.P. Gent. Frozen development in graph coloring. *Theoretical Computer Science*, 2001. To appear. Available from `www.apes.cs.strath.ac.uk/apesreports.html`.

[Gent and Walsh, 1998] I.P. Gent and T. Walsh. Analysis of heuristics for number partitioning. *Computational Intelligence*, 14(3):430–451, 1998.

[Gomes *et al.*, 1998] C. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *The 4th Int. Conf. on Artificial Intelligence Planning Systems (AIPS'98)*, 1998.

[Kamath *et al.*, 1995] A. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. *Randomized Structure and Algorithms*, 7:59–80, 1995.

[Kirkpatrick and Toulouse, 1985] S. Kirkpatrick and G. Toulouse. Configuration space analysis of the traveling salesman problem. *J. de Physique*, 46:1277–1292, 1985.

[Korf, 1995] R. Korf. From approximate to optimal solutions: A case study of number partitioning. In *Proc. of the 14th IJCAI*. 1995.

[Mitchell *et al.*, 1992] D. Mitchell, B. Selman, and H. Levesque. Hard and Easy Distributions of SAT Problems. In *Proc. of the 10th Nat. 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 Nat. Conf. on AI*, pages 340–345. 1997.

[Schneider *et al.*, 1996] J. Schneider, C. Froschhammer, I. Morgernstern, T. Husslein, and J.M. Singer. Searching for backbones – an efficient parallel algorithm for the traveling salesman problem. *Comput. Phys. Commun.*, 96:173–188, 1996.

[Singer *et al.*, 2000a] J. Singer, I.P. Gent, and A. Smaill. Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research*, 12:235–270, 2000.

[Singer *et al.*, 2000b] J. Singer, I.P. Gent, and A. Smaill. Local search on random 2+p-SAT. In *Proc. of the 14th ECAI*. 2000.

[Slaney and Thiébaux, 1996] J. Slaney and S. Thiébaux. Linear time near-optimal planning in the blocks world. In *Proc. of 13th Nat. Conf. on AI*, pages 1208–1214. 1996.

[Slaney and Thiébaux, 1998] J. Slaney and S. Thiébaux. On the hardness of decision and optimisation problems. In *Proc. of the 13th ECAI*, pages 244–248. 1998.

Cooperative Search and Nogood Recording

Cyril Terrioux
LSIS - Equipe INCA (LIM)
39, rue Joliot-Curie
F-13453 Marseille Cedex 13 (France)
e-mail: terrioux@lim.univ-mrs.fr

Abstract

Within the framework of constraint satisfaction problem, we propose a new scheme of cooperative parallel search. The cooperation is realized by exchanging nogoods (instantiations which can't be extended to a solution). We associate a process with each solver and we introduce a manager of nogoods, in order to regulate exchanges of nogoods. Each solver runs the algorithm Forward-Checking with Nogood Recording. We add to algorithm a phase of interpretation, which limits the size of the search tree according to the received nogoods. Solvers differ from each other in ordering variables and/or values by using different heuristics. The interest of our approach is shown experimentally. In particular, we obtain linear or superlinear speed-up for consistent problems, like for inconsistent ones, up to about ten solvers.

1 Introduction

In constraint satisfaction problems, one of main tasks consists in determining whether there exists a solution, i.e. an instantiation of all variables which satisfies all constraints. This task is a NP-complete problem. In order to speed up the resolution of problems, parallel searches are used. A basic one is *independent parallel search* which consists in running several solvers (each one using a different heuristic) instead of a single solver. The aim is that at least one of the solvers gets an heuristic suitable for the problem which we solve. Tested on the graph coloring problem ([Hogg and Williams, 1994]), this approach has better results than a classical resolution with a single solver, but the gains seem limited. Hogg and Williams recommend then the use of a cooperative parallel search.
A *cooperative parallel search* is based on the same ideas as the independent search with in addition an exchange of informations between solvers, in order to guide solvers to a solution, and then, to speed up the resolution. Experimental results on cryptarithmetic problems ([Clearwater *et al.*, 1991; Hogg and Huberman, 1993]) and on graph coloring problem ([Hogg and Huberman, 1993; Hogg and Williams, 1993]) show a significant gain in time with respect to an independent search. In both cases, the exchanged informations correspond to partial consistent instantiations.

In [Martinez and Verfaillie, 1996], a cooperation based on exchanging nogoods (i.e. instantiations which can't be extended to a solution) is proposed. Exchanged nogoods permit solvers to prune their own search tree. So, one can expect to find more quickly a solution. Solvers run the algorithm Forward Checking with Nogood Recording (noted FC-NR [Schiex and Verfaillie, 1993]). The realized implementation gathers all solvers in a single process which simulates the parallelism. It is turned to a monoprocessor system. Experimentations on random CSPs show that cooperative search is better than independent one. However, the weak gain with report to a single solver gives a doubt about efficiency of a such search.

From the idea of Martinez and Verfaillie, we define a new scheme of cooperation with exchange of nogoods, turned to systems with one or several processors. We associate a process with each solver. Each solver runs FC-NR. In order to avoid problems raised by the cost of communications, we introduce a manager of nogoods whose role is to regulate exchange of nogoods. In addition to sending and receipt of messages, we add a phase of interpretation to FC-NR, in order to limit the size of the search tree according to received nogoods.
Our second main aim is to answer an open question ([Martinez and Verfaillie, 1996]) about efficiency of a cooperative parallel search with exchange of nogoods. We show experimentally the interest of our approach.

The plan is as follows. In section 2, we give basic notions about CSPs, nogoods and FC-NR. Then, in section 3, we present our scheme by describing the manager of nogoods and the phase of interpretation. Finally, after providing experimental results in section 4, we conclude in section 5.

2 Definitions

2.1 Definitions about CSPs

A *constraint satisfaction problem* (CSP) is defined by a quadruplet (X, D, C, R). X is a set $\{x_1, \ldots, x_n\}$ of n variables. Each variable x_i takes its values in the domain D_i from D. Variables are subject to constraints from C. Each constraint c involves a set $X_c = \{x_{c_1}, \ldots, x_{c_k}\}$ of variables. A relation R_c (from R) is associated with each constraint c such that R_c represents the set of allowed k-uplets over $D_{c_1} \times \cdots \times D_{c_k}$.
A CSP is called *binary* if each constraint involves two vari-

ables. Let x_i and x_j be two variables, we note c_{ij} the corresponding constraint. Afterwards, we consider only binary CSPs. However, our ideas can be extended to n-ary CSPs.

Given $Y \subseteq X$ such that $Y = \{x_1, \ldots, x_k\}$, an *instantiation* of variables from Y is a k-uplet $(v_1, \ldots, v_k)$ from $D_1 \times \cdots \times D_k$. It is called *consistent* if $\forall c \in C, X_c \subseteq Y, (v_1, \ldots, v_k)[X_c] \in R_c$, *inconsistent* otherwise. We use indifferently the term *assignment* instead of instantiation. We note the instantiation $(v_1, \ldots, v_k)$ in the more meaningful form $(x_1 \leftarrow v_1, \ldots, x_k \leftarrow v_k)$. A *solution* is a consistent instantiation of all variables. Given an instance $\mathcal{P} = (X, D, C, R)$, determine whether $\mathcal{P}$ has a solution is a NP-complete problem.

Given a CSP $\mathcal{P} = (X, D, C, R)$ and an instantiation $\mathcal{A}_i = \{x_1 \leftarrow v_1, x_2 \leftarrow v_2, \ldots, x_i \leftarrow v_i\}$, $\mathcal{P}(\mathcal{A}_i) = (X, D(\mathcal{A}_i), C, R(\mathcal{A}_i))$ is the *CSP induced* by $\mathcal{A}_i$ from $\mathcal{P}$ with a Forward Checking filter such that:

- $\forall j, 1 \leq j \leq i, D_j(\mathcal{A}_i) = \{v_j\}$
- $\forall j, i < j \leq n, D_j(\mathcal{A}_i) = \{v_j \in D_j | \forall c_{kj} \in C, 1 \leq k \leq i, (v_k, v_j) \in R_{c_{kj}}\}$
- $\forall j, j', R_{c_{jj'}}(\mathcal{A}_i) = R_{c_{jj'}} \cap (D_j(\mathcal{A}_i) \times D_{j'}(\mathcal{A}_i))$.

$\mathcal{A}_i$ is said *FC-consistent* if $\forall j, D_j(\mathcal{A}_i) \neq \emptyset$.

2.2 Nogoods: definitions and properties

In this part, we give the main definitions and properties about nogoods and FC-NR ([Schiex and Verfaillie, 1993]).
A nogood corresponds to an assignment which can't be extended to a solution. More formally ([Schiex and Verfaillie, 1993]), given an instantiation $\mathcal{A}$ and a subset J of constraints ($J \subseteq C$), $(\mathcal{A}, J)$ is a *nogood* if the CSP (X, D, J, R) doesn't have any solution which contains $\mathcal{A}$. J is called the nogood's *justification* (we note X_J the variables subject to constraints from J). The *arity* of the nogood $(\mathcal{A}, J)$ is the number of assigned variables in $\mathcal{A}$. We note $\mathcal{A}[Y]$ the restriction of $\mathcal{A}$ to variables which are both in $X_\mathcal{A}$ and in Y.
For instance, every inconsistent assignment corresponds to a nogood. The converse doesn't hold.

To calculate justifications of nogoods, we use the notion of "value-killer" (introduced in [Schiex and Verfaillie, 1993]) and we extend it in order to exploit it in our scheme. Given an assignment $\mathcal{A}_i$, the CSP $\mathcal{P}(\mathcal{A}_i)$ induced by $\mathcal{A}_i$, and the set N of nogoods found by other solvers, a constraint c_{kj} ($j > i \geq k$) is a *value-killer* of value v_j from D_j for $\mathcal{A}_i$ if one of the following conditions holds:

1. c_{kj} is a value-killer of v_j for $\mathcal{A}_{i-1}$
2. $k = i$ and $(v_k, v_j) \notin R_{c_{kj}}(\mathcal{A}_i)$ and $v_j \in D_j(\mathcal{A}_{i-1})$
3. $\{x_k \leftarrow v_k, x_j \leftarrow v_j\} \in N$

If a unique solver is used, $N = \emptyset$ (which corresponds to the definition presented in [Schiex and Verfaillie, 1993]).

Assume that an inconsistency is detected because a domain D_i becomes empty. The reasons of failure (i.e. justifications) correspond to the union of value-killers of D_i. The following theorem formalizes the creation of nogoods from dead-ends.

Theorem 1 *Let $\mathcal{A}$ be an assignment and x_i be an unassigned variable. Let K be the set of value-killers of D_i. If it doesn't remain any value in $D_i(\mathcal{A})$, then $(\mathcal{A}[X_K], K)$ is a nogood.*

The two next theorems make it possible to create new nogoods from existing nogoods. The first theorem builds a new nogood from a single existing nogood.

Theorem 2 (projection [Schiex and Verfaillie, 1993]) *If $(\mathcal{A}, J)$ is a nogood, then $(\mathcal{A}[X_J], J)$ is a nogood.*

In other words, we keep from instantiation the variables which are involved in the inconsistency. Thus, we produce a new nogood whose arity is limited to its strict minimum.
Theorem 3, we build a new nogood from a set of nogoods:

Theorem 3 *Let $\mathcal{A}$ be an instantiation, x_i be an unassigned variable. Let K be the set of value-killers of D_i. Let $\mathcal{A}_j$ be the extension of $\mathcal{A}$ by assigning the value v_j to x_i ($\mathcal{A}_j = \mathcal{A} \cup \{x_i \leftarrow v_j\}$). If $(\mathcal{A}_1, J_1), \ldots, (\mathcal{A}_d, J_d)$ are nogoods, then $(\mathcal{A}, K \cup \bigcup_{j=1}^{d} J_j)$ is a nogood.*

A nogood can be used either to backjump or to add a new constraint or to tighten an existing constraint. In both cases, it follows from the use of nogoods a pruning of the search tree.

FC-NR explores the search tree like Forward Checking, During the search, it takes advantage of dead-ends to create and record nogoods. These nogoods are then used as described above to prune the search tree. The main drawback of FC-NR is that the number of nogoods is potentially exponential. So, we limit the number of nogoods by recording nogoods whose arity is at most 2 (i.e. unary or binary nogoods), according to the proposition of Schiex and Verfaillie ([Schiex and Verfaillie, 1993]). Nevertheless, the ideas we present can be easily extended to n-ary nogoods.

3 Description of our multiple solver

Our multiple solver consists of p sequential solvers which run independently FC-NR on the same CSP. Each solver differs from another one in ordering variables and/or values with different heuristics. Thus, each one has a different search tree. The cooperation consists in exchanging nogoods. A solver can use nogoods produced by other solvers in order to prune a part of its search tree, which should speed up the resolution.

During the search, solvers produce nogoods which are communicated to other solvers. Therefore, when a solver finds a nogood, it must send $p-1$ messages to inform its partners. Although the number of nogoods is bounded, the cost of communications can become very important, prohibitive even. So, we add a process called "manager of nogoods", whose role is to inform solvers of the existence of nogoods. Accordingly, when a solver finds a nogood, it informs the manager, which communicates at once this new information to a part of other solvers. In this way, a solver sends only one message and gets back more quickly to the resolution of the problem.
The next paragraph is devoted to the role and the contribution of manager in our scheme.

3.1 The manager of nogoods

Role of the manager

The manager's task is to update the base of nogoods and to communicate new nogoods to solvers. Update the base of

nogoods consists in adding constraints to initial problem or in tightening the existing constraints. To a unary (respectively binary) nogood corresponds a unary (resp. binary) constraint. Each nogood communicated to manager is added to the base.

In order to limit the cost of communications, the manager must inform only solvers for which nogoods may be useful. A nogood is said *useful* for a solver if it allows this solver to limit the size of its search tree.
The next theorem characterizes the usefulness of a nogood according to its arity and the current instantiation.

Theorem 4 (characterization of the usefulness)
(a) a unary nogood is always useful,

(b) a binary nogood $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ *is useful if, in the current instantiation,* x_i *and* x_j *are assigned respectively to* a *and* b*,*

(c) a binary nogood $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ *is useful if, in the current instantiation,* x_i *(resp.* x_j*) is assigned to* a *(resp.* b*),* x_j *(resp.* x_i*) isn't assigned and* b *(resp.* a*) isn't removed yet.*

Proof: see [Terrioux, 2001].

From this theorem, we explicite what solvers receive some nogoods (according to their usefulness):

(a) every unary nogood is communicated to all solvers (except the solver which finds it),

(b) binary nogood $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ is communicated to each solver (except the solver which finds it) whose instantiation contains $x_i \leftarrow a$ or $x_j \leftarrow b$.

In case (b), we can't certify that the nogood is useful, because the solver may have backtracked between the sending of the nogood by the manager and its receipt by the solver.
With a view to limit the cost of communications, only the instantiation $\mathcal{A}$ of nogood $(\mathcal{A}, J)$ is conveyed. Communicate the justification isn't necessary because this nogood is added to the problem in the form of a constraint c. Thanks to received information, solvers can forbid $\mathcal{A}$ with justification c.

Contribution of the manager
In this part, we show the contribution of the manager of nogoods to our scheme with respect to a version without manager. The comparison is based on the total number of messages which are exchanged during all search.
Let N be the total number of nogoods which are exchanged by all solvers. We count U unary nogoods and B binary ones. Note that, among these N nogoods, doubles may exist. In effect, two solvers can find independently a same nogood.
In a scheme without manager, each solver communicates the nogoods it finds to $p-1$ other solvers. So, $U(p-1)$ messages are sent for unary nogoods and $B(p-1)$ for binary ones. But, in a scheme with manager, nogoods are first sent to manager by solvers. During all search, solvers convey to manager U messages for unary nogoods and B for binary ones. Then, the manager sends only u unary nogoods to $p-1$ solvers. These u nogoods correspond to U nogoods minus the doubles. Likewise, for binary nogoods, doubles aren't communicated. Furthermore, for the remaining binary nogoods, the manager restricts the number of recipients. Let b be the number of messages sent by manager for binary nogoods. In our scheme, we exchange $U + u(p-1)$ messages for unary nogoods and $B + b$ messages for binary ones.

In the worst case, the scheme with manager produces up to N additional messages in comparison with the scheme without manager. But, in general, u and b are little enough so that the scheme with manager produces fewer messages.

3.2 Phase of interpretation

The method which we are going to describe is applied whenever a nogood is received. Solvers check whether a message is received after developing a node and before filtering.
In the phase of interpretation, solvers analyze received nogoods in order to limit the size of their search tree by stopping branch which can't lead to solution or by enforcing additional filtering. For unary nogoods, this phase corresponds to a permanent deletion of a value and to a possible backjump. Method 1 details the phase for such nogoods.

Method 1 (phase of interpretation for unary nogoods)
Let $\mathcal{A}$ *be the current instantiation. Let* $(\{x_i \leftarrow a\}, J)$ *be the received nogood.*
We delete a *from* D_i*.*

(a) If x_i *is assigned to the value* a*, then we backjump to* x_i*. If* D_i *is empty, we record the nogood* $(\mathcal{A}[X_K], K)$*, with* K *the set of value-killers of* D_i*.*

(b) If x_i *is assigned to* b *(*$b \neq a$*), we do nothing.*

(c) If x_i *isn't assigned, we check whether* D_i *is empty. If* D_i *is empty, we record the nogood* $(\mathcal{A}[X_K], K)$*, with* K *the set of value-killers of* D_i*.*

Theorem 5 *The method 1 is correct.*

Proof: see [Terrioux, 2001].

For binary nogoods, the phase corresponds to enforce an additional filtering and to a possible backjump. Method 2 describes the actions done during this phase for binary nogoods.

Method 2 (phase of interpretation for binary nogoods)
Let $\mathcal{A}$ *be the current instantiation and* $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ *be the received nogood.*

(a) If x_i *and* x_j *are assigned in* $\mathcal{A}$ *to* a *and* b *respectively, then we backjump to the deepest variable among* x_i *and* x_j*. If* x_j *(resp.* x_i*) is this variable, we delete by filtering* b *(resp.* a*) from* D_j *(resp.* D_i*).*

(b) If x_i *(resp.* x_j*) is assigned to* a *(resp.* b*) and* x_j *(resp.* x_i*) isn't be assigned, we delete by filtering* b *(resp.* a*) from* D_j *(resp.* D_i*).*

If D_j *(resp.* D_i*) becomes empty, we record the nogood* $(\mathcal{A}[X_K], K)$ *with* K *the set of value-killers of* D_j *(resp.* D_i*).*

Theorem 6 *The method 2 is correct.*

Proof: see [Terrioux, 2001].

Unlike the phase of interpretation for unary nogoods, here, the deletion isn't permanent.
Whereas the phase of interpretation is correct, its addition to FC-NR may, in some cases, compromise a basic property of FC-NR. The next paragraph is devoted to this problem.

3.3 Maintening FC-consistency

We remind first a basic property of FC-NR (from FC):

Property 1 *Every instantiation built by FC-NR is FC-consistent.*

After a backtrack to the variable x_i, FC-NR (like FC) cancels the filtering which follows the assignment of x_i. So, it restores each domain in its previous state. In particular, if a domain is wiped out after filtering, it isn't empty after restoring. It ensues the preserve of the property 1.
With the addition of communications and of the phase of interpretation, this property may be compromised. For example, we consider the search tree explored by a solver which cooperates with other ones. Let $D_4 = \{a, b, c, d\}$. This solver assignes first a to x_1, then b to x_2. Enforce FC-consistency after assigning x_2 removes a from D_4. The filtering which follows the assignment of c_1 to x_3 deletes b and c from D_4. The solver assignes d to x_4, and then, it visits the corresponding subtree. Assume that this subtree contains only failures and that the solver receives unary nogoods which forbid assigning values b and c to x_4. So the solver backtracks and records a unary nogood which forbids d for x_4. It backtracks again (to x_3) and assignes c_2 to x_3, which raises a problem, namely D_4 is empty (due to permanent removals of b, c and d from D_4 by unary nogoods). So, the current instantiation isn't FC-consistent and the property 1 doesn't hold.
The next theorem characterizes the problem:

Theorem 7
Let $\mathcal{A}_j = \{x_1 \leftarrow v_1, \ldots, x_{j-1} \leftarrow v_{j-1}, x_j \leftarrow r\}$ be a FC-consistent instantiation. We consider the exploration by FC-NR of subtree rooted in $x_j \leftarrow r$. Let NG be the set of values of x_i which remain forbidden by nogoods at the end of the exploration such that these nogoods are recorded or received during this exploration and none of them doesn't involve x_j. If all values of $D_i(\mathcal{A}_j)$ are removed during the exploration, no value is restored in $D_i(\mathcal{A}_{j-1})$ after cancelling the filtering following the assignment of r to x_j if and only if $D_i(\mathcal{A}_{j-1}) \subseteq NG$.

Proof: see [Terrioux, 2001].
It ensues that, in some cases, the union of receipts and creations of nogoods with the filtering induces the existence of empty domains, and thus property 1 doesn't hold.

A solution consists in checking whether a domain is wiped out after cancelling the last filtering and, if there exists such domain, in backtracking until the domain isn't empty. It isn't necessary to check the emptiness of every domain, thanks to the following lemma which determines potential origins of this problem.

Lemma 1 *Only recording or receipt of a nogood may induce the loss of property 1.*

Proof: see [Terrioux, 2001].
This lemma enables to check only the emptiness of domains which become empty after recording or receiving a nogood. If emptiness is induced by receiving a nogood, we introduce an additional phase of backjump. This phase follows every detection of empty domain after receiving a nogood and makes it possible to keep on with the search from a FC-consistent instantiation.

Method 3 (backjump's phase)
If D_i is empty due to a received nogood, we backjump:

- *until D_i isn't empty or until the current instantiation is empty, if the nogood is unary,*
- *until D_i isn't empty, if the nogood is binary.*

Note that we backtrack to empty instantiation only for inconsistent problems.

Theorem 8 *The method 3 is correct.*

Proof: see [Terrioux, 2001].

If emptiness is induced by recording a nogood, a solution consists in recording only nogoods which don't wipe out a domain. However, we communicate all nogoods to manager. This solution is easy to implement, but it doesn't take advantage of all found nogoods.

4 Experimental results

4.1 Experimental protocol

We work on random instances produced by random generator written by D. Frost, C. Bessière, R. Dechter and J.-C. Régin. This generator takes 4 parameters N, D, C and T. It builds a CSP of class (N, D, C, T) with N variables which have domains of size D and C binary constraints ($0 \leq C \leq \frac{N(N-1)}{2}$) in which T tuples are forbidden ($0 \leq T \leq D^2$).

Experimental results we give afterwards concern classes $(50, 25, 123, T)$ with T which varies between 433 and 447. Considered classes are near to the satisfiability's threshold which corresponds to $T = 437$. However, for FC-NR, the difficulty's shape is observed for $T = 439$. Every problem we consider has a connected graph of constraints.

Given results are the averages of results obtained on 100 problems per class. Each problem is solved 15 times in order to reduce the impact of non-determinism of solvers on results. Results of a problem are then the averages of results of 15 resolutions. For a given resolution, the results we consider are ones of the solver which solves the problem first.

Experimentations are realized on a Linux-based PC with an Intel Pentium III 550 MHz processor and 256 Mb of memory.

4.2 Heuristics

In order to guarantee distinct search trees, each solver orders variables and/or values with different heuristics. As there exist few efficient heuristics, from an efficient heuristic H for choosing variables, we produce several different orders by choosing differently the first variable and then applying H.
We use the heuristic dom/deg ([Bessière and Régin, 1996]), for which the next variable to assign is one which minimizes the ratio $\frac{|D_i|}{|\Gamma_i|}$ (where D_i is the current domain of x_i and Γ_i is the set of variables which are connected to x_i by a binary constraint). This heuristic is considered better, in general, than other classical heuristics. That's why we choose it.
In our implementation, only the size of domains varies for each instantiation. The degree $|\Gamma_i|$ is updated when a new constraint is added thanks to a nogood.
As regards the choice of next value to assign, we consider values in appearance order or in reverse order. In following results (unless otherwise specified), half solvers use the appearance order to order domains, other half reverse order.

T	# consistent Problems	p							
		2	4	6	8	10	12	14	16
433	76	151.632	145.088	135.303	128.465	110.872	98.330	92.450	86.654
434	70	133.537	143.058	135.602	136.825	128.594	123.678	112.790	97.157
435	66	144.337	145.998	131.788	134.207	121.954	122.111	110.581	106.031
436	62	157.517	138.239	135.508	119.489	110.208	101.167	96.752	91.180
437	61	114.282	138.451	135.791	126.069	120.020	108.559	102.327	90.007
438	37	139.957	153.035	149.573	135.236	133.701	119.713	106.998	96.255
439	39	129.954	127.950	113.481	120.610	107.390	96.568	88.068	79.107
440	31	124.797	127.585	114.503	109.981	100.412	93.810	90.037	82.020
441	25	134.811	133.188	131.791	122.755	113.251	102.487	93.784	87.489
442	13	105.809	136.557	123.936	118.738	105.576	96.864	85.484	75.888
443	10	146.562	131.673	120.268	113.724	100.108	90.449	82.566	75.646
444	8	135.528	137.906	126.939	118.453	107.629	97.878	88.722	77.433
445	3	139.624	122.885	116.870	107.777	99.315	89.736	81.375	73.706
446	2	125.955	127.126	117.049	108.319	98.783	89.387	79.130	73.316
447	3	144.414	132.838	116.935	106.912	94.329	84.449	74.738	65.866

Table 1: Efficiency (in %) for consistent and inconsistent problems for classes $(50, 25, 123, T)$.

4.3 Results

Efficiency

In this paragraph, we assess the speed-up and the efficiency of our method. Let T_1 be the run-time of a single solver for the resolution of a serie of problems and T_p be the run-time for p solvers which are run in parallel. We define speed-up as the ratio $\frac{T_1}{T_p}$ and efficiency as the ratio $\frac{T_1}{p\,T_p}$. The speed-up is called linear with report to the number p of solvers if it equals to p, superlinear if it is greater than p, sublinear otherwise.
Table 1 presents efficiency (in %) obtained for classes $(50, 25, 123, T)$ with T which varies between 433 and 447. In table 1, up to 10 solvers, we obtain linear or superlinear speed-up for all classes (except 3 classes). Above 10 solvers, some classes have linear or superlinear speed-up, but most classes have sublinear speed-up. We note also a decrease of efficiency with the increase of number of solvers.

According to consistency of problems, we observe a better speed-up for consistent problems (which remains linear or superlinear). We note the same decrease of efficiency with the number of solvers. But, efficiency for inconsistent problems is less important, whereas it is greater than 1 up to 10 solvers. It follows the appearance of sublinear speed-up above 10 solvers. This lack of efficiency for inconsistent problems infers a decrease of efficiency for overall problems.

Explanations of obtained results

First, we take an interest in explaining observed gains. Possible origns are multiple orders of variables and cooperation. We define an independent version of our scheme (i.e. without exchange of nogoods). We compare the two versions by calculating the ratio of the run-time for the independent version over one for cooperative version. Figure 1 presents results obtained for the class (50,25,123,439) with a number of solvers between 2 and 8. We observe similar results for other classes. We note first that cooperative version is always better than independent one. Then, we observe that the ratio is near 1 for consistent problems (solid line). That means that the good quality of results, for these problems, results mostly from multiple orders of variables. However, the ratio remains

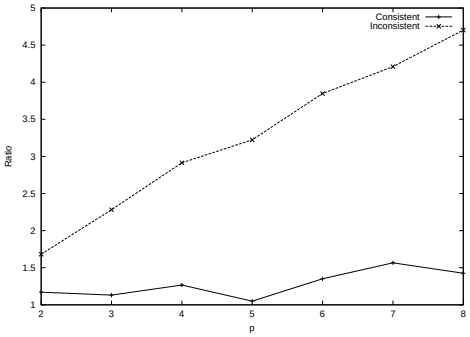

Figure 1: Ratio independent search / cooperative search.

greater than 1. So, exchange of nogoods participates in obtained gains too.
Finally, for inconsistent problems (dashed line), ratio is more important than for consistent ones. It increases with the number of solvers. In other words, obtained results for these problems are mostly due to cooperation and the contribution of cooperation increases with the number of solvers.
For inconsistent problems, we must underline the predominant role of values heuristics. For each solver s (except one if the number of solvers is odd), there exists a solver which uses the same variables heuristic as s and the values heuristic which is reverse with report to one of s. Without exchanging nogoods, these two solvers visit similar search trees. With exchanging nogoods, each one explores only a part of its search tree thanks to received nogoods. It's the same for consistent problems, but this effect is less important because the search stops as soon as a solution is found.

We focus then on possible reasons of efficiency's decrease. With a scheme like our, an usual reason of efficiency's lack is the importance of cost of communications. Our method doesn't make exception. But, in our case, there is another reason which explains the decrease of performances.
We compare multiple solvers S_8 and S'_8. Both have 8 solvers. For ordering values, half solvers of S_8 use the appearance or-

der, other half the reverse order. All solvers of S'_8 use the appearance order. We realise that S_8 has a better efficiency than S'_8. The number of messages for S'_8 is greater than for S_8. But, above all, it's the same for the number of nodes. So S'_8 explores more important trees. S_8 and S'_8 differ in used heuristics for ordering values and variables. The heuristics we use for ordering variables are near each other. Using two different orders of values adds diversity to resolution. Thus, S_8 is more various than S'_8. This difference of diversity permits to explain the gap of efficiency between S_8 and S'_8.
The lack of diversity is the main reason (with the increase of number of communications) of the efficiency's decrease.

Number of messages and real contribution of manager
In order to measure the real contribution of manager, we compare the costs of communications in a scheme with manager and one without manager. In presented results, we consider that the cost of a message is independent of presence or not of the manager, and that the communication of a binary nogood is twice as expensive as one of a unary nogood (because a binary nogood consists of two pairs variable-value, against a single pair for a unary nogood). Figure 2 presents the ratio of cost of communications for a scheme without manager over one for a scheme with manager. Considered problems (consistent and inconsistent) belong to class (50,25,123,439). For information, we observe similar results for other classes.

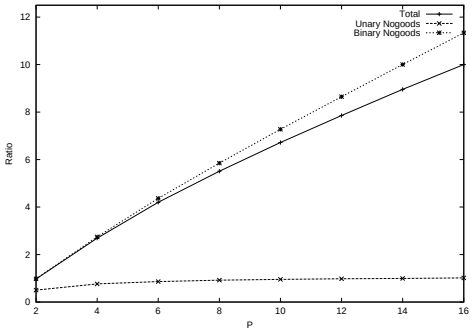

Figure 2: Ratio between schemes with and without manager.

First, we note an increase of manager's contribution with p. Thus, we can hope that the number of solvers above of which the cost of communications penalizes efficiency is greater in a scheme with manager than one without manager.

More precisely, we note that communications of unary nogoods is more expensive in the scheme with manager, when p is less important. This result was foreseeable. Indeed, in case of unary nogoods, the manager removes only doubles. As the probability that two solvers find simultaneously the same nogoods is all smaller since there are few solvers. We explain thus that the scheme with manager becomes better than one without manager when the number of solvers increases.
Regarding the cost of communications for binary nogoods, the scheme with manager is significantly cheaper and this economy increases with p. This result is explained by the fact that binary nogoods are, in general, useful for few solvers.
On overall communications, the scheme with manager is the best, due essentially to the number of binary nogoods which is significantly greater than one of unary nogoods (with a factor between 30 and 100).

In conclusion, the manager does its job by limiting the number of exchanged messages. It avoids solvers a lack of time due to management of messages (in particular, receipt of useless nogoods).

5 Conclusions and future works

In this paper, from an idea of Martinez and Verfaillie, we define a new scheme of cooperative parallel search by exchanging nogoods and we assess experimentally its efficiency. We observe then linear or superlinear speed-up up to 10 solvers for inconsistent problems and up to 16 solvers for consistent ones. So, exchange of nogoods is an efficient form of cooperation. We note a decrease of efficiency with the number of solvers, due to the increasing number of communications and to a lack of diversity of solvers.

A first extension of this work consists in finding several efficient and diverse heuristics in order to improve efficiency as well as increase the number of solvers. Then, we can extend our scheme by applying any algorithm which maintains some level of consistency, by using different algorithms (which would permit to combine complete search methods and incomplete ones like in [Hogg and Williams, 1993] and to improve the diversity of solvers), or by generalizing it to another form of cooperation with exchange of informations.

References

[Bessière and Régin, 1996] C. Bessière and J.-C. Régin. MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proc. of CP 96*, pages 61–75, 1996.

[Clearwater *et al.*, 1991] S. Clearwater, B. Huberman, and T. Hogg. Cooperative Solution of Constraint Satisfaction Problems. *Science*, 254:1181–1183, Nov. 1991.

[Hogg and Huberman, 1993] T. Hogg and B. A. Huberman. Better Than the Best: The Power of Cooperation. In L. Nadel and D. Stein, editors, *1992 Lectures in Complex Systems*, volume V of *SFI Studies in the Sciences of Complexity*, pages 165–184. Addison-Wesley, 1993.

[Hogg and Williams, 1993] T. Hogg and C.P. Williams. Solving the Really Hard Problems with Cooperative Search. In *Proc. of AAAI 93*, pages 231–236, 1993.

[Hogg and Williams, 1994] T. Hogg and C.P. Williams. Expected Gains from Parallelizing Constraint Solving for Hard Problems. In *Proc. of AAAI 94*, pages 331–336, Seattle, WA, 1994.

[Martinez and Verfaillie, 1996] D. Martinez and G. Verfaillie. Echange de Nogoods pour la résolution coopérative de problèmes de satisfaction de contraintes. In *Proc. of CNPC 96*, pages 261–274, Dijon, France, 1996. In french.

[Schiex and Verfaillie, 1993] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. In *Proc. of the* 5^{th} *IEEE ICTAI*, 1993.

[Terrioux, 2001] C. Terrioux. Cooperative search and nogood recording. Research report, LIM, 2001.

Search on High Degree Graphs

Toby Walsh
Department of Computer Science
University of York
York
England
tw@cs.york.ac.uk

Abstract

We show that nodes of high degree tend to occur infrequently in random graphs but frequently in a wide variety of graphs associated with real world search problems. We then study some alternative models for randomly generating graphs which have been proposed to give more realistic topologies. For example, we show that Watts and Strogatz's small world model has a narrow distribution of node degree. On the other hand, Barabási and Albert's power law model, gives graphs with both nodes of high degree and a small world topology. These graphs may therefore be useful for benchmarking. We then measure the impact of nodes of high degree and a small world topology on the cost of coloring graphs. The long tail in search costs observed with small world graphs disappears when these graphs are also constructed to contain nodes of high degree. We conjecture that this is a result of the small size of their "backbone", pairs of edges that are frozen to be the same color.

1 Introduction

How does the topology of graphs met in practice differ from uniform random graphs? This is an important question since common topological structures may have a large impact on problem hardness and may be exploitable. Barabási and Albert have shown that graphs derived from areas as diverse as the World Wide Web, and electricity distribution contain more nodes of high degree than are likely in random graphs of the same size and edge density [Barabási and Albert, 1999]. As a second example, Redner has shown that the citation graph of papers in the ISI catalog contains a few nodes of very high degree [Render, 1998]. Whilst 633,391 out of the 783,339 papers receive less than 10 citations, 64 are cited more than 1000 times, and one received 8907 citations. The presence of nodes with high degree may have a significant impact on search problems. For instance, if the constraint graph of a scheduling problem has several nodes with high degree, then it may be difficult to solve as some resources are scarce. As a second example, if the adjacency graph in a Hamiltonian circuit problem has many nodes of high degree, then the problem may be easy since there are many paths into and out of these nodes, and it is hard to get stuck at a "dead-end" node. Search heuristics like Brelaz's graph coloring heuristic [Brelaz, 1979] are designed to exploit such variation in node degree.

This paper is structured as follows. We first show that nodes of high degree tend to occur infrequently in random graphs but frequently in a wide variety of real world search problems. As test cases, we use exactly the same problems studied in [Walsh, 1999]. We then study some alternative models for randomly generating graphs which give more non-uniform graphs (specifically Barabási and Albert's power law model, Watts and Strogatz's small world model, and Hogg's ultrametric model). Finally, we explore the impact of nodes of high degree on search and in particular, on graph coloring algorithms.

2 Random graphs

Two types of random graphs are commonly used, the $G_{n,m}$ and the $G_{n,p}$ models. In the $G_{n,m}$ model, graphs with n nodes and m edges are generated by sampling uniformly from the $n(n-1)/2$ possible edges. In the $G_{n,p}$ model, graphs with n nodes and an expected number of $pn(n-1)/2$ edges are generated by including each of the $n(n-1)/2$ possible edges with fixed probability p. The two models have very similar properties, including similar distributions in the degree of nodes. In a random $G_{n,p}$ graph, the probability that a node is directly connected to exactly k others, $p(k)$ follows a Poisson distribution. More precisely,

$$p(k) = e^{-\lambda}\lambda^k/k!$$

where n is the number of nodes, p is the probability that any pair of nodes are connected, and λ is $(n-1)p$, the expected node degree. As the Poisson distribution decays exponentially, nodes of high degree are unlikely.

In this paper, we focus on the cumulative probability, $P(k)$ which is the probability of a node being directly connected to k or less nodes:

$$P(k) = \sum_{i=1}^{k} p(i).$$

Whilst $p(k)$ is smoothly varying for random graphs, it can behave more erratically on real world graphs. The cumulative probability, which is by definition monotonically increasing, tends to give a clearer picture. Figure 1 shows that the cumu-

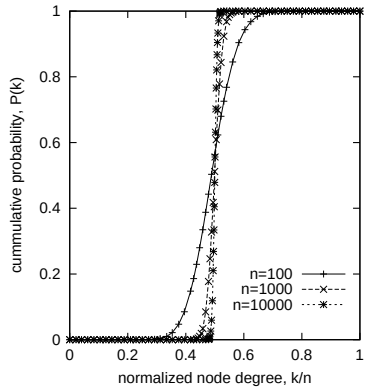

Figure 1: Cumulative probability (y-axis) against the normalized node degree (x-axis) for random $G_{n,p}$ graphs with $p = 0.5$.

lative probability against the normalized degree for random graphs rapidly approaches a step function as n increases. The degree of nodes therefore becomes tightly clustered around the average degree.

3 Real world graphs

We next studied the distribution in the degree of nodes found in the real world graphs studied in [Walsh, 1999].

3.1 Graph coloring

We looked at some real world graph coloring problems from the DIMACS benchmark library. We focused on the register allocation problems as these are based on real program code. Figure 2 demonstrates that these problems have a very

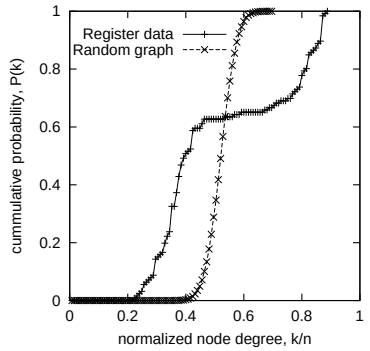

Figure 2: Cummulative probability (y-axis) against the normalized node degree (x-axis). "Register data" is the `zeronin.i.1` register allocation problem which is converted into a graph coloring problem with 125 nodes and 4100 edges. "Random graph" is a random graph of the same size and edge density. Other problems in the DIMACS graph coloring benchmark gave similar results.

skewed distribution in the degree of their nodes. Other problems from the DIMACS benchmark library gave very similar cumulative probability distributions for the degree of their nodes. Compared to random graphs of the same size and edge density, these register allocation problems have a number of nodes that are of much higher and lower degree than the average. For example, the node of maximum degree in Figure 2 is directly connected to 89% of the nodes in the graph. This is more than twice the average degree, and there is less than a 1 in 4 million chance that a node in a random graph of the same size and edge density has degree as large as this. On the other hand, the node of least degree has less than half the average degree, and there is less than a 1 in 7 million chance that a node in a random graph of the same size and edge density has degree as small as this. The plateau region in the middle of the graph indicates that there are very few nodes with the average degree. Most nodes have either higher or lower degrees. By comparison, the degrees of nodes in a random graph are tightly clustered around the average. A similar plateau region around the average degree is seen in most of the register allocation problems in the DIMACS benchmark library.

3.2 Time-tabling

Time-tabling problems can be naturally modelled as graph coloring problems, with classes represented by nodes and time-slots by colors. We therefore tested some real world time-tabling problems from the Industrial Engineering archive at the University of Toronto. Figure 3 demonstrates that problems in this dataset also have a skewed distribution in the degree of their nodes. Other benchmark problems from this library gave very similar curves. Compared to random graphs with the same number of nodes and edges, these time-tabling problems have a number of nodes that have much higher and lower degree than the average. For example, the node of maximum degree in Figure 3 is directly connected to 71% of the nodes in the graph. This is nearly three times the average degree, and there is less than a 1 in 10^{20} chance that a node in a random graph of the same size and edge density has degree as large as this. On the other hand, the node of least degree has approximately one tenth of the average degree, and there is less than a 1 in 10^{15} chance that a node in a random graph of the same size and edge density has degree as small as this. [Walsh, 1999] suggests that sparse problems in this dataset have more clustering of nodes than the dense problems. However, there was no obvious indication of this in the distribution of node degrees.

3.3 Quasigroups

A quasigroup is a Latin square, a m by m multiplication table in which each entry appears just once in each row or column. Quasigroups can model a variety of practical problems like sports tournament scheduling and the design of factorial experiments. A number of open questions in finite mathematics about the existence (or non-existence) of quasigroups with particular properties have been answered using model finding and constraint satisfaction programs [Fujita *et al.*, 1993]. Recently, a class of quasigroup problems have been proposed as a benchmark for generating hard and satisfiable problem instances for local search methods [Achlioptas *et al.*, 2000].

An order m quasigroup problem can be represented as a binary constraint satisfaction problem with m^2 variables, each

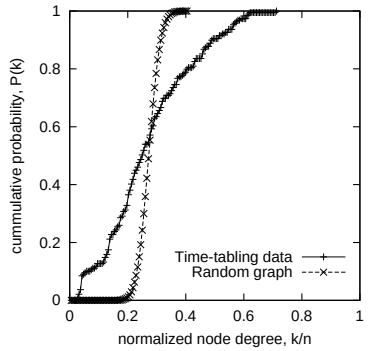

Figure 3: Cummulative probability (y-axis) against the normalized node degree (x-axis). "Time-tabling data" is the Earl Haig Collegiate time-tabling problem which is converted into a graph coloring problem with 188 nodes and 4864 edges. "Random graph" is a random graph of the same size and edge density. Other problems from the Industrial Engineering archive at the University of Toronto gave similar results.

with a domain of size m. The constraint graph for such a problem consists of $2m$ cliques, one for each row and column, with each clique being of size m. Each node in the constraint graph is connected to $2(m-1)$ other nodes. Hence, $p(k) = 1$ if $k = 2(m-1)$ and 0 otherwise, and the cumulative probability $P(k)$ is a step function at $k = 2(m-1)$. As all nodes in the constraint graph of a quasigroup have the same degree, quasigroups may suffer from limitations as a benchmark. For example, the Brelaz heuristic [Brelaz, 1979] (which tries to exploit variations in the degree of nodes in the constraint graph) may perform less well on quasigroup problems than on more realistic benchmarks in which there is a variability in the degree of nodes.

4 Non-uniform random models

As the $G_{n.m}$ and $G_{n,p}$ models tend to give graphs with a narrow distribution in the degree of nodes, are there any better models for randomly generating graphs? In this section, we look at three different random models, all proposed by their authors to give more realistic graphs.

4.1 Small world model

Watts and Strogatz showed that graphs that occur in many biological, social and man-made systems are often neither completely regular nor completely random, but have instead a "small world" topology in which nodes are highly clustered yet the path length between them is small [Watts and Strogatz, 1998]. Such graphs tend to occur frequently in real world search problems [Walsh, 1999]. To generate graphs with a small world topology, we randomly rewire a regular graph like a ring lattice [Watts and Strogatz, 1998; Gent *et al.*, 1999]. The ring lattice provides nodes that are highly clustering, whilst the random rewiring introduces short cuts which rapidly reduces the average path length. Unfortunately, graphs constructed in this manner tend not to have a wide distribution in the degree of nodes, and in particular are unlikely to contain any nodes of high degree. For small amounts of rewiring, $p(k)$ peaks around the lattice degree, and converges on the Poisson distribution found in random graphs for more extensive rewiring.

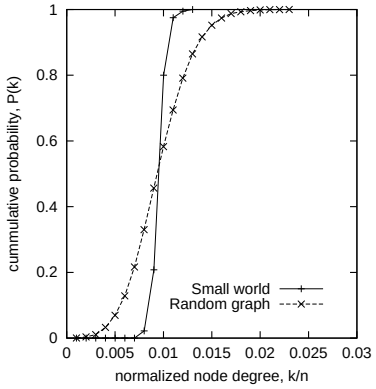

Figure 4: Cummulative probability (y-axis) against the normalized node degree (x-axis). "Small world" is a graph with a small world topology generated by randomly rewiring a ring lattice of 1000 nodes, each with 10 neighbors with a rewiring probability, $p = 1/16$. "Random graph" is a random graph of the same size and edge density.

In Figure 4, we plot the cumulative probability for the node degrees of graphs generated to have a small world topology by randomly rewiring a ring lattice. Small world graphs have a distribution of node degrees that is narrower than that for random graphs with the same number of nodes and edges. Due to the lack of variability in the degree of nodes, these small world graphs may have limitations as a model of real world graphs. The absence of nodes of high degree is likely to impact on search performance. For instance, heuristics like Brelaz which try to exploit variations in node degree are likely to find these graphs harder to color than graphs with a wider variability in node degree. Can we find a model with a variability in the node degree that is similar to that seen in the real world graphs studied in the previous section?

4.2 Ultrametric model

To generate graphs with more realistic structures, Hogg has proposed a model based on grouping the nodes into a tree-like structure [Hogg, 1996]. In this model, an ultrametric distance between the n nodes is defined by grouping them into a binary tree and measuring the distance up this tree to a common ancestor. A pair of nodes at ultrametric distance d is joined by an edge with relative probability p^d. If $p = 1$, graphs are purely random. If $p < 1$, graphs have a hierarchical clustering as edges are more likely between nearby nodes. Figure 5 gives the cumulative probability distribution for the node degrees in a graph generated with an ultrametric distance using the model from [Hogg, 1996]. There is a definite broadening of the distribution in node degrees compared to random graphs. Nodes of degree higher and lower than the average occur more frequently in these ultrametric graphs than in random graphs. For example, one node in the ultrametric graph

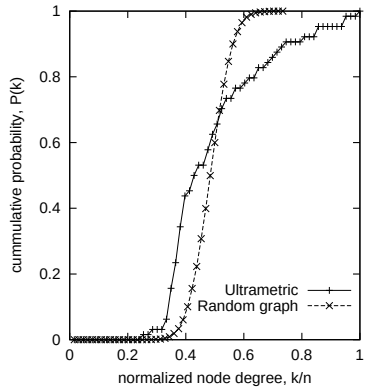

Figure 5: Cummulative probability (y-axis) against the normalized node degree (x-axis). "Ultrametric" is a graph with a ultrametric world topology generated with 64 nodes, 1008 edges (to give an average degree of $n/2$) and $p = 1/4$. "Random graph" is a random graph of the same size and edge density.

is connected to all the other nodes. This node has more than twice the average degree, and there is less than a 1 in 3 million chance that a node in a random graph of the same size and edge density has degree as large as this. On the other hand, the node of least degree has just over half the average degree, and there is less than a 1 in 500 chance that a node in a random graph of the same size and edge density has degree as small as this. Ultrametric graphs thus provide a better model of the distribution of node degrees. However, they lack a small world topology as nodes are not highly clustered [Walsh, 1999]. Can we find a model which has both a small world topology (which has shown to be common in real world graphs) and a large variability in the node degree (which has also been shown to be common)?

4.3 Power law model

Barabási and Albert have shown that real world graphs containing nodes of high degree often follow a power law in which the probability $p(k)$ that a node is connected to k others is proportional to $k^{-\gamma}$ where γ is some constant (typically around 3) [Barabási and Albert, 1999]. Redner has shown that highly cited papers tend to follow a Zipf power law with exponent approximately -1/2 [Render, 1998]. It follows from this result that the degree of nodes in the citation graph for highly cited papers follows a power law with $p(k)$ proportional to k^{-3}. Such power law decay compares to the exponential decay in $p(k)$ seen in random graphs.

To generate power law graphs, Barabási and Albert propose a model in which, starting with a small number of nodes (n_0), they repeatedly add new nodes with m ($m \leq n_0$) edges. These edges are preferentially attached to nodes with high degree. They suggest a linear model in which the probability that an edge is attached to a node i is $k_i / \sum_j k_j$ where k_j is the degree of node j. Using a mean-field theory [Barabási *et al.*, 1999], they show that such a graph with n nodes has:

$$p(k) = \frac{2m^2(n - n_0)}{n} \frac{1}{k^3}$$

That is, $p(k)$ is proportional to $k^{-\gamma}$ where $\gamma = 3$. Note that $p(k)$ is also proportional to m^2, the square of the average degree of the graph. In the limit of large n, $p(k) \mapsto \frac{2m^2}{k^3}$. The presence of non-linear terms in the preferential attachment probability will change the nature of this power law scaling and may be a route to power laws in which the scaling exponent is different to 3.

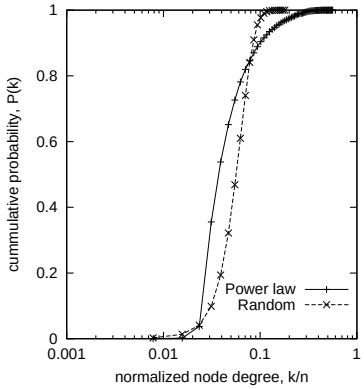

Figure 6: Cummulative probability (y-axis) against the normalized node degree (x-axis) for graphs generated to have a simple power law scaling in their node degree. Note the logscale used for the x-axis. "Power law" is a graph constructed by the modified Barabási and Albert's model with $m_0 = 1$, $m = 16$ and $n = 128$; "Random" is a random graph of the same size and edge density.

We propose a minor modification to this model to tackle the problem that the average degree m is bounded by the size of the initial graph n_0. This will hinder the construction of high density graphs (which were not uncommon in the previous section). We suggest connecting an edge to a node i with probability $\min(1, mk_i / \sum_i k_i)$. Each new node is then connected to the graph by approximately m edges on average. This modification is similar to moving from the $G_{n.m}$ to the $G_{n.p}$ model of random graphs.

In Figure 6, we plot the cumulative probability for the degree of nodes in graphs generated by this modified model. As with the ultrametric graphs, we observe a definite broadening of the distribution in node degrees compared to random graphs. Nodes of degree higher and lower than the average occur more frequently in these power law graphs than in random graphs. For example, the node of maximum degree is directly connected to 70% of the nodes in the graph. This is more than three times the average degree, and there is less than a 1 in 10^{18} chance that a node in a random graph of the same size and edge density has degree as large as this. On the other hand, the node of least degree has nearly one fifth of the average degree, and there is less than a 1 in 10^7 chance that a node in a random graph of the same size and edge density

has degree as small as this. Unlike random graphs in which the distribution sharpens as we increase the size of graphs, we see a similar spread in the distribution of node degrees as these graphs are increased in size.

Ideally, we want like a method for generating graphs that gives graphs with both nodes of high degree and a small world topology. The nodes of high degree generated by the (modified) Barabási and Albert model are likely to keep the average path length short. But are the nodes likely to be tightly clustered? Table 1 demonstrates that these graphs tend to have a small world topology as the graph size is increased.

n	L	L_{rand}	C	C_{rand}	μ
16	1.00	1.00	1.00	1.00	1.00
32	1.24	1.24	0.81	0.77	1.05
64	1.57	1.56	0.57	0.43	1.35
128	1.77	1.78	0.39	0.24	1.62
256	1.93	1.89	0.25	0.12	2.12
512	2.07	2.10	0.16	0.06	2.58

Table 1: Average path lengths (L) and clustering coefficients (C) for graphs constructed to display a simple power law in the node degree. The clustering coefficient is the average fraction of neighbors directly connected to each other and is a measure of "cliqueness". Graphs have n nodes and are generated by the modified Barabási and Albert model using $n_0 = 1$ and $m = 16$. For comparison, the characteristic path lengths (L_{rand}) and clustering coefficients (C_{rand}) for random graphs of the same size and edge density are also given. The last column is the proximity ratio (μ), the normalized ratio of the clustering coefficient and the characteristic path length (i.e. C/C_{rand} / L/L_{rand}). Graphs with a proximity ratio, $\mu > 1$ have a small world topology.

5 Search

Graphs generated by the modified Barabási and Albert model have both a broad distribution in degree of their nodes and a small world topology. These are both features which are common in real world graphs but rare in random graphs. These graphs may therefore be good benchmarks for testing graph coloring algorithms. They may also be useful for benchmarking other search problems involving graphs (e.g. for generating the constraint graph in constraint satisfaction problems, the adjacency graph in Hamiltonian circuit problems, ...)

Unfortunately coloring graphs generated by the (modified) Barabási and Albert model is typically easy. Most heuristics based on node degree can quickly (in many cases, immediately) find a $m + 1$-coloring. In addition, a m-clique can be quickly found within the nodes of high degree showing that a $m + 1$-coloring is optimal. A simple fix to this problem is to start with an initial graph which is not a clique. This initial graph could be a ring lattice as in [Watts and Strogatz, 1998; Walsh, 1999], or the inter-linking constraint graph of a quasigroup as in [Gent *et al.*, 1999]. In both cases, we observe similar results. The choice of the initial graph has little effect on the evolution of nodes of high degree. In addition, starting from a ring lattice or the constraint graph of a quasigroup promotes the appearance of a small world topology. As in [Achlioptas *et al.*, 2000; Gent *et al.*, 1999], we generate problems with a mixture of regular structure (from the initial graph) and randomness (from the addition of new nodes).

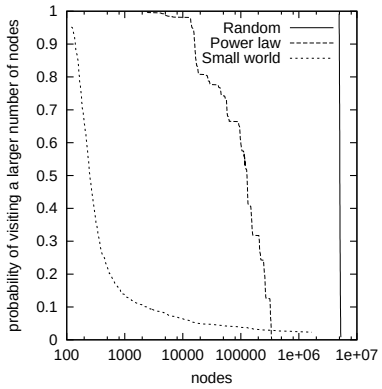

Figure 7: Number of search nodes (x-axis) against probability of visiting this many search nodes (y-axis) when coloring graphs generated to have either a power law scaling in their node degree, a small world topology or a purely random topology. Note the logscale used for the x-axis. "Power law" is a 125 node graph constructed by the modified Barabási and Albert's model, starting from the constraint graph of an order 5 quasigroup, adding additional nodes into the graph with 10 edges each on average; "Random" is a random graph of the same size and edge density; "Small world" is a graph formed by randomly rewiring a 125 node ring lattice, each node starting with 10 neighbours, and each edge being rewired with probability 1/16. Other instances of power law, random and small world graphs generated with the same parameters gave similar search cost distributions.

In Figure 7, we plot the distribution in search costs for coloring graphs with either a power law scaling in their node degree, a small world topology or a purely random topology. To find optimal colorings, we use an algorithm due to Mike Trick which is based upon Brelaz's DSATUR algorithm [Brelaz, 1979]. Unlike small world graphs, power law graphs do not display a long tail in the distribution of search costs. Whilst power law graphs are easier to color than random graphs, there is a larger spread in search costs for power law graphs than for random graphs. The absence of a long tail means that there are less benefits with these power law graphs for a randomization and rapid restart strategy [Gomes *et al.*, 1997; 1998] compared to small world graphs [Walsh, 1999].

5.1 Backbones

Recent efforts to understand the hardness of satisfiability problems has focused on "backbone" variables that are frozen to a particular value in all solutions [Monasson *et al.*, 1998]. It has been shown, for example, that hard random 3-SAT problems from the phase transition have a very large backbone [Parkes, 1997]. Backbone variables may lead to thrashing behaviour since search algorithms can branch incorrectly

on them. If these branching mistakes occur high in the search tree, they can be very costly to undo. The idea of backbone variable has been generalized to graph coloring [Culberson and Gent, 2000]. Since any permutation of the colors is also a valid coloring, we cannot look at nodes which must take a given color. Instead, we look at nodes that cannot be colored differently. As in [Culberson and Gent, 2000], two nodes are **frozen** in a k-colorable graph if they have the same color in all valid k-colorings. No edge can occur between two nodes that are frozen. The **backbone** is simply the set of frozen pairs.

The power law graphs generated by the modified Barabási and Albert model in Figure 7 had very small backbones. Indeed, in many cases, there are only one or two pairs of nodes in the backbone. At the start of search, it is therefore hard to color incorrectly any of the nodes in one of these power law graphs. This helps explain the lack of a long tail in the distribution of search costs. By comparison, the small world graphs had backbones with between fifty and one hundred pairs of nodes in them. At the start of search, it is therefore easy to color incorrectly one of nodes. This gives rise to a long tail in the distribution of search costs for backtracking algorithms like Brelaz's DSATUR algorithm.

6 Conclusions

We have shown that nodes of high degree tend to occur infrequently in random graphs but frequently in a wide variety of real world search problems. As test cases, we used exactly the problem studied in [Walsh, 1999]. We then studied some alternative models for randomly generating non-uniform graphs. Watts and Strogatz's small world model gives graphs with a very narrow distribution in node degree, whilst Hogg's ultrametric model gives graphs containing nodes of high degree but lacks a small world topology. Barabási and Albert's power law model combines the best of both models, giving graphs with nodes of high degree and with a small world topology. Such graphs may be useful for benchmarking graph coloring, constraint satisfaction and other search problems involving graphs. We measured the impact of both nodes of high degree and a small world topology on a graph coloring algorithm. The long tail in search costs observed with small world graphs disappears when these graphs are also constructed to contain nodes of high degree. This may be connected to the small size of their "backbone", pairs of edges frozen with the same color.

What general lessons can be learnt from this research? First, search problems met in practice may be neither completely structured nor completely random. Since algorithms optimized for purely random problems may perform poorly on problems that contain both structure and randomness, it may be useful to benchmark with problem generators that introduce both structure and randomness. Second, in addition to a small world topology, many real world graphs display a wide variation in the degree of their nodes. In particular, nodes of high degree occur much more frequently than in purely random graphs. Third, these simple topological features can have a major impact on the cost of solving search problems. We conjecture that graph coloring heuristics like Brelaz are often able to exploit the distribution in node degree, preventing much of thrashing behaviour seen in more uniform graphs.

Acknowledgements

The author is an EPSRC advanced research fellow and wishes to thank the other members of the APES research group.

References

[Achlioptas *et al.*, 2000] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proc. of 17th Nat. Conf. on AI*. 2000.

[Barabási and Albert, 1999] A-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[Barabási *et al.*, 1999] A-L. Barabási, R. Albert, and H. Jeong. Mean-field theory for scale-free random networks. *Physica A*, 272:173–187, 1999.

[Brelaz, 1979] D. Brelaz. New methods to color the vertices of a graph. *Communications of ACM*, 22:251–256, 1979.

[Culberson and Gent, 2000] J. Culberson and I.P. Gent. Frozen development in graph coloring. *Theoretical Computer Science*, 2001. To appear.

[Fujita *et al.*, 1993] Masayuki Fujita, John Slaney, and Frank Bennett. Automatic generation of some results in finite algebra. In *Proc. of the 13th IJCAI*, pages 52–57. 1993.

[Gent *et al.*, 1999] I.P. Gent, H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proc. of the 16th Nat. Conf. on AI*. 1999.

[Gomes *et al.*, 1997] C. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In G. Smolka, editor, *Proc. of 3rd Int. Conf. on Principles and Practice of Constraint Programming (CP97)*, pages 121–135. Springer, 1997.

[Gomes *et al.*, 1998] C. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *The 4th International Conference on Artificial Intelligence Planning Systems (AIPS'98)*, 1998.

[Hogg, 1996] T. Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, 81(1–2):127–154, 1996.

[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 Nat. Conf. on AI*, pages 340–345. 1997.

[Render, 1998] S. Render. How popular is your paper? An empirical study of the citation distribution. *European Physical Journal B*, 4:131–134, 1998.

[Walsh, 1999] T. Walsh. Search in a small world. In *Proceedings of 16th IJCAI*. Artificial Intelligence, 1999.

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

SEARCH, SATISFIABILITY, AND CONSTRAINT SATISFACTION PROBLEMS

SATISFIABILITY

Backjumping for Quantified Boolean Logic Satisfiability

Enrico Giunchiglia and **Massimo Narizzano** and **Armando Tacchella**
DIST - Università di Genova
Viale Causa 13, 16145 Genova, Italy
{enrico,mox,tac}@mrg.dist.unige.it

Abstract

The implementation of effective reasoning tools for deciding the satisfiability of Quantified Boolean Formulas (QBFs) is an important research issue in Artificial Intelligence. Many decision procedures have been proposed in the last few years, most of them based on the Davis, Logemann, Loveland procedure (DLL) for propositional satisfiability (SAT). In this paper we show how it is possible to extend the conflict-directed backjumping schema for SAT to QBF: when applicable, it allows to jump over existentially quantified literals while backtracking. We introduce solution-directed backjumping, which allows the same for universally quantified literals. Then, we show how it is possible to incorporate both conflict-directed and solution-directed backjumping in a DLL-based decision procedure for QBF satisfiability. We also implement and test the procedure: The experimental analysis shows that, because of backjumping, significant speed-ups can be obtained. While there have been several proposals for backjumping in SAT, this is the first time –as far as we know– this idea has been proposed, implemented and experimented for QBFs.

1 Introduction

The implementation of effective reasoning tools for deciding the satisfiability of Quantified Boolean Formulas (QBFs) is an important research issue in Artificial Intelligence. Many reasoning tasks involving abduction, reasoning about knowledge, non monotonic reasoning, are PSPACE-complete reasoning problems and are reducible in polynomial time to the problem of determining the satisfaction of a QBF. More important, since QBF reasoning is the prototypical PSPACE problem, many of these reductions are readily available. For these reasons, we have seen in the last few years the presentation of several implemented decision procedures for QBFs, like QKN [Kleine-Büning, H. and Karpinski, M. and Flögel, A., 1995], EVALUATE [Cadoli *et al.*, 1998, Cadoli *et al.*, 2000], DECIDE [Rintanen, 1999b], QUIP [Egly *et al.*, 2000], QSOLVE [Feldmann *et al.*, 2000]. Most of the above decision procedures are based on the Davis, Logemann, Loveland procedure (DLL) for propositional satisfiability [Davis *et al.*, 1962] (SAT). This is because it is rather easy to extend DLL to deal with QBFs, and also because DLL is at the core of many state-of-the-art deciders for SAT.

In this paper we show how it is possible to extend the conflict-directed backjumping schema for SAT (see, e.g., [Prosser, 1993, Bayardo, Jr. and Schrag, 1997]) to QBF: when applicable, it allows to jump over existentially quantified literals while backtracking. We introduce solution-directed backjumping, which allows the same for universally quantified literals. Then, we show how it is possible to incorporate both conflict-directed and solution-directed backjumping in a DLL-based decision procedure for QBF satisfiability. We also implement and test the procedure: The experimental analysis shows that, because of backjumping, significant speed-ups can be obtained. While there have been several proposals for backjumping in SAT, this is the first time –as far as we know– this idea has been proposed, implemented and experimented for QBFs.

The paper is structured as follows. In Section 2 we introduce some formal preliminaries necessary for the rest of the paper. In Section 3 we present QUBE, a DLL based decision procedure for QBFs. Section 4 is devoted to the presentation of the theoretical results at the basis of the backjumping procedure presented in Section 5. The experimental analysis is reported in Section 6. We end the paper with the conclusions and future work in Section 7. Proofs are omitted for lack of space.

2 Formal preliminaries

Consider a set P of propositional letters. An *atom* is an element of P. A *literal* is an atom or the negation of an atom. In the following, for any literal l,

- $|l|$ is the atom occurring in l; and
- $\bar{l}$ is $\neg l$ if l is an atom, and is $|l|$ otherwise.

A *clause* C is an n-ary ($n \geq 0$) disjunction of literals such that, for any two distinct disjuncts l, l' in C, it is not the case that $|l| = |l'|$. A *propositional formula* is a k-ary ($k \geq 0$) conjunction of clauses. As customary in SAT, we represent a clause as a set of literals, and a propositional formula as a set of clauses. With this notation, e.g.,

- the clause $\{\}$ is the *empty clause* and stands for the empty disjunction,
- the propositional formula $\{\}$ is the *empty set of clauses* and stands for the empty conjunction,
- the propositional formula $\{\{\}\}$ stands for the set of clauses whose only element is the empty clause.

A *QBF* is an expression of the form

$$Q_1 x_1 \dots Q_n x_n \Phi \qquad (n \geq 0) \tag{1}$$

where

- every Q_i $(1 \leq i \leq n)$ is a quantifier, either existential $\exists$ or universal $\forall$,
- $x_1, \dots, x_n$ are pairwise distinct atoms in P, and
- Φ is a propositional formula in the atoms $x_1, \dots, x_n$.

$Q_1 x_1 \dots Q_n x_n$ is the *prefix* and Φ is the *matrix* of (1).[1] For example, the expression

$$\begin{aligned}\forall x_1 \exists x_2 \forall x_3 \exists x_4 \exists x_5 \{&\{x_1, x_3, x_4\}, \{x_1, \neg x_3, \neg x_4, \neg x_5\},\\ &\{x_1, \neg x_4, x_5\}, \{\neg x_1, x_2, x_5\},\\ &\{\neg x_1, x_3, x_4\}, \{\neg x_1, x_3, \neg x_4\},\\ &\{\neg x_1, \neg x_2, \neg x_3, \neg x_5\}\}\end{aligned} \tag{2}$$

is a QBF with 7 clauses.

Consider a QBF (1). A literal l is

- *existential* if $\exists|l|$ belongs to the prefix of (1), and is *universal* otherwise.
- *unit* in (1) if l is existential, and, for some $m \geq 0$,
  - a clause $\{l, l_1, \dots, l_m\}$ belongs to Φ, and
  - each expression $\forall|l_i|$ $(1 \leq i \leq m)$ occurs at the right of $\exists|l|$ in the prefix of (1).
- *monotone* if either l is existential, l occurs in Φ, and $\overline{l}$ does not occur in Φ; or l is universal, l does not occur in Φ, and $\overline{l}$ occurs in Φ.

A clause C is *contradictory* if no existential literal belongs to C. For example, the empty clause is contradictory.

The semantics of a QBF φ can be defined recursively as follows. If φ contains a contradictory clause then φ is FALSE. If the matrix of φ is the empty set of clauses then φ is TRUE. If φ is $\exists x \psi$ (respectively $\forall x \psi$), φ is TRUE if and only if φ_x or (respectively and) $\varphi_{\neg x}$ are TRUE. If $\varphi = Q x \psi$ is a QBF and l is a literal, φ_l is the QBF obtained from ψ by deleting the clauses in which l occurs, and removing $\overline{l}$ from the others. It is easy to see that if φ is a QBF without universal quantifiers, the problem of deciding φ satisfiability reduces to SAT.

3 QUBE

QUBE is implemented in C on top of SIM, an efficient decider for SAT developed by our group [Giunchiglia *et al.*, 2001a]. A high-level description of QUBE is presented in Figure 1.[2] In Figure 1,

- φ is a global variable initially set to the input QBF.

```
1  φ := ⟨the input QBF⟩;     Stack := ⟨the empty stack⟩;
2  function Simplify()
3    do
4      φ' := φ;
5      if (⟨a contradictory clause is in φ⟩) return FALSE;
6      if (⟨the matrix of φ is empty⟩) return TRUE;
7      if (⟨l is unit in φ⟩)
8        |l|.mode := UNIT; Extend(l);
9      if (⟨l is monotone in φ⟩)
10       |l|.mode := PURE; Extend(l);
11   while (φ' ≠ φ);
12   return UNDEF;
13 function Backtrack(res)
14   while (⟨Stack is not empty⟩)
15     l := Retract();
16     if (|l|.mode = L-SPLIT) and
17       ((res = FALSE and |l|.type = ∃) or
18        (res = TRUE and |l|.type = ∀))
19         |l|.mode := R-SPLIT; return l̄;
20   return NULL;
21 function QubeSolver()
22   do
23     res := Simplify();
24     if (res = UNDEF)  l := ChooseLiteral();
25     else  l := Backtrack(res);
26     if (l ≠ NULL)  Extend(l);
27   while (l ≠ NULL);
28   return res;
```

Figure 1: The algorithm of QUBE.

- *Stack* is a global variable storing the search stack, and is initially empty.
- FALSE, TRUE, UNDEF, NULL, UNIT, PURE, L-SPLIT, R-SPLIT are pairwise distinct constants.
- for each atom x in the input QBF,
  - $x.mode$ is a property of x whose possible values are UNIT, PURE, L-SPLIT, R-SPLIT, and have the obvious meaning, and
  - $x.type$ is $\exists$ if x is existential, and $\forall$ otherwise.
- *Extend*(l) deletes the clauses of φ in which l occurs, and removes $\overline{l}$ from the others. Additionally, before performing the above operations, pushes l and φ in the stack.
- *Retract*() pops the literal and corresponding QBF that are on top of the stack: the literal is returned, while the QBF is assigned to φ. (Intuitively, *Retract* is the "inverse" operation of *Extend*).
- *Simplify*() simplifies φ till a contradictory clause is generated (line 5), or the matrix of φ is empty (line 6),

[1] Traditionally, the syntax of QBFs allows for arbitrary propositional formulas as matrices. However, the problem of determining the satisfiability of a QBF whose matrix is a set of clauses remains PSPACE complete.

[2] We use the following pseudocode conventions. Indentation indicates block structure. Two instructions on the same line belong to the same block. ":=" is the assignment operator. The constructs **while** ⟨*cond*⟩ ⟨*block*⟩, **do** ⟨*block*⟩ **while** ⟨*cond*⟩, **if**⟨*cond*⟩ ⟨$block_1$⟩ **else** ⟨$block_2$⟩ have the same interpretation as in the C language.

or no simplification is possible (lines 4, 11). The simplifications performed in lines 8 and 10 correspond to Lemmas 6, 4-5 respectively of [Cadoli *et al.*, 1998, Cadoli *et al.*, 2000].

- *ChooseLiteral*() returns a literal l occurring in φ such that for each atom x occurring to the left of $|l|$ in the prefix of the input QBF,
  - x and $\neg x$ do not occur in φ, or
  - x is existential iff l is existential.

  ChooseLiteral() also sets $|l|.mode$ to L-SPLIT.
- *Backtrack*(*res*): pops all the literals and corresponding QBFs (line 15) from the stack, till a literal l is reached such that $|l|.mode$ is L-SPLIT (line 16), and either
  - l is existential and *res* = FALSE (line 17); or
  - l is universal and *res* = TRUE (line 18).

  If such a literal l exists, $|l|.mode$ is set to R-SPLIT, and $\overline{l}$ is returned (line 19). If no such literal exists, NULL is returned (line 20).

It is easy to see that QUBE is a generalization of DLL: QUBE and DLL have the same behavior on QBFs without universal quantifiers.

To understand QUBE behavior, consider the QBF (2). For simplicity, assume that *ChooseLiteral* returns the negation of the first atom in the prefix which occurs in the matrix of the QBF under consideration. Then, the tree searched by QUBE when φ is (2) is represented in Figure 2. In Figure 2, each node shows

- the sequence of literals assigned by QUBE before a branch takes place (first line): for each literal l in the sequence, we also show the value of $|l|.mode$; and
- the matrix of the resulting QBF, prefixed by a label (second line).

As the result of the computation, QUBE would correctly return FALSE, i.e., (2) is unsatisfiable.

4 Backjumping

Let φ be a QBF (1). Consider φ.

In the following, for any finite sequence $\mu = l_1; \ldots; l_m$ $(m \geq 0)$ of literals, we write

- *Assign*(μ, Φ) as an abbreviation for

$$\textit{Assign}(l_m, \ldots, (\textit{Assign}(l_1, \Phi)) \ldots),$$

  where, if Ψ is a set of clauses, *Assign*(l, Ψ) is the set of clauses obtained from Ψ by deleting the clauses in which l occurs, and removing $\overline{l}$ from the others.
- $[Q_1 x_1 \ldots Q_n x_n]_\mu$ as an abbreviation for the expression obtained from $Q_1 x_1 \ldots Q_n x_n$ by removing $Q_i x_i$ whenever x_i or $\neg x_i$ is in μ.
- φ_μ as an abbreviation for the QBF

$$[Q_1 x_1 \ldots Q_n x_n]_\mu \textit{Assign}(\mu, \Phi).$$

Intuitively, if μ is a sequence of literals representing an assignment, φ_μ is the QBF resulting after the literals in μ are assigned. For example, considering Figure 2, if φ is (2), Φ is $(F4)$, and μ is $\neg x_1; x_3$, then φ_μ is $\exists x_2 \exists x_4 \exists x_5 \Phi$.

As in constraint satisfaction (see, e.g., [Dechter, 1990, Prosser, 1993, Bayardo, Jr. and Schrag, 1997]) it may be the case that only a subset of the literals in the current assignment is responsible for the result (either TRUE or FALSE) of φ_μ satisfiability. Then, assuming that it is possible to effectively determine such a subset ν, we could avoid doing a right branch on a literal l, if l is not in ν. To make these notions precise we need the following definitions.

A finite sequence $\mu = l_1; \ldots; l_m$ $(m \geq 0)$ of literals is an *assignment for* φ if for each literal l_i in μ

- l_i is unit, or monotone in $\varphi_{l_1;\ldots;l_{i-1}}$; or
- l_i occurs in $\varphi_{l_1;\ldots;l_{i-1}}$ and for each atom x occurring to the left of $|l_i|$ in the prefix of the input QBF,
  - x and $\neg x$ do not occur in $\varphi_{l_1;\ldots;l_{i-1}}$, or
  - x is existential iff l_i is existential.

Consider an assignment $\mu = l_1; \ldots; l_m$ for φ.
A set of literals ν is a *reason for* φ_μ *result* if

- $\nu \subseteq \{l_1, \ldots, l_m\}$, and
- for any assignment μ' for φ such that
  - $\nu \subseteq \{l : l \text{ is in } \mu'\}$,
  - $\{|l| : l \text{ is in } \mu'\} = \{|l| : l \text{ is in } \mu\}$,

  $\varphi_{\mu'}$ is satisfiable iff φ_μ is satisfiable.

For example, if φ is (2),

- if μ is $x_1; \neg x_2$, then $\{x_1\}$ is a reason for φ_μ result: For each assignment $\mu' \in \{x_1; \neg x_2, x_1; x_2\}$, $\varphi_{\mu'}$ is unsatisfiable, and
- if μ is $\neg x_1; \neg x_3$, then $\{\neg x_1\}$ is a reason for φ_μ result: For each assignment $\mu' \in \{\neg x_1; \neg x_3, \neg x_1; x_3\}$, $\varphi_{\mu'}$ is satisfiable.

Intuitively,

- when given an assignment μ such that either the matrix φ_μ is empty or it contains a contradictory clause, we first compute a reason ν for φ_μ result, and
- while backtracking, we dynamically modify the reason ν for the current result. Furthermore, we use ν in order to avoid useless branches: In particular, we avoid doing a right branch on a literal l if l is not in ν.

The theorems in the next subsections —in which we show how to compute the reason— are an easy consequence of the following proposition.

Let ν be a reason for φ_μ result. We say that

- ν is a *reason for* φ_μ *satisfiability* if φ_μ is satisfiable, and
- ν is a *reason for* φ_μ *unsatisfiability*, otherwise.

Proposition 1 *Let φ be a QBF (1). Let $\mu = l_1; \ldots; l_m$ $(m \geq 0)$ be an assignment for φ. Let μ' be a sequence obtained by removing some of the literals in μ. Let ν be the set of literals in μ'.*

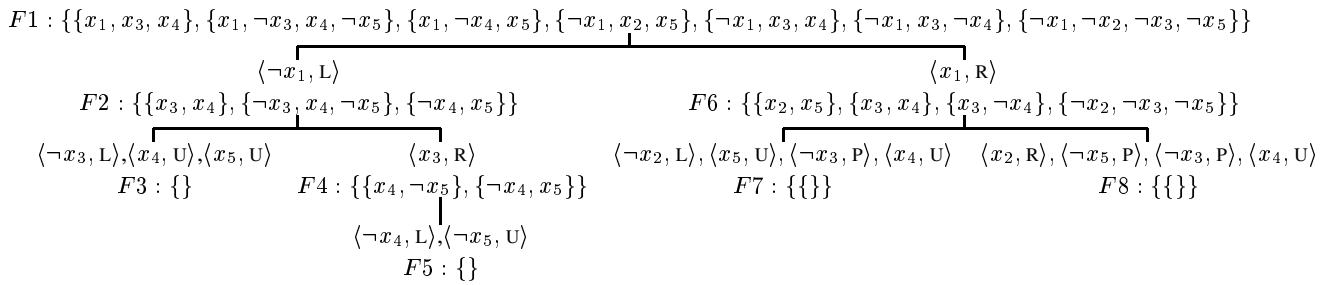

Figure 2: QuBE computation tree for (2). U, P, L, R stand for UNIT, PURE, L-SPLIT, R-SPLIT respectively. The prefix is $\forall x_1 \exists x_2 \forall x_3 \exists x_4 \exists x_5$.

- *ν is a reason for φ_μ unsatisfiability iff the QBF*

$$\exists |l_1| \ldots \exists |l_m| [Q_1 x_1 \ldots Q_n x_n]_\mu \mathit{Assign}(\mu', \Phi)$$

is unsatisfiable.

- *ν is a reason for φ_μ satisfiability iff the QBF*

$$\forall |l_1| \ldots \forall |l_m| [Q_1 x_1 \ldots Q_n x_n]_\mu \mathit{Assign}(\mu', \Phi)$$

is satisfiable.

4.1 Conflict-directed Backjumping

The following Theorem allows us to compute the reason for φ_μ result when the matrix of φ_μ contains a contradictory clause. Given a sequence of literals μ, we say that a literal l is FALSE-*irrelevant* in μ if —assuming μ' is the sequence obtained by deleting l in μ— $\varphi_{\mu'}$ contains a contradictory clause.

Theorem 1 *Let φ be a QBF. Let μ be an assignment for φ such that φ_μ contains a contradictory clause. Let μ' be a sequence obtained by recursively eliminating a* FALSE*-irrelevant literal in μ. Let ν be the set of literals in μ'. Then ν is a reason for φ_μ unsatisfiability.*

If φ is (2), this Theorem allows us to conclude, e.g., that —with reference to Figure 2—

- if μ is $x_1; \neg x_2; x_5; \neg x_3; x_4$, then $\{x_1, \neg x_3, x_4\}$ is a reason for φ_μ unsatisfiability, and
- if μ is $x_1; \neg x_2; x_5; \neg x_3; \neg x_4$, then $\{x_1, \neg x_3, \neg x_4\}$ is a reason for φ_μ unsatisfiability.

Our next step is to show how it is possible to compute reasons for φ_μ unsatisfiability while backtracking.

Theorem 2 *Let φ be a QBF. Let l be a literal. Let $\mu; l$ be an assignment for φ. Let ν be a reason for $\varphi_{\mu;l}$ unsatisfiability.*

1. *If l is not in ν, then ν is a reason for φ_μ unsatisfiability.*
2. *If $l \in \nu$, and l is universal, then $\nu \setminus \{l\}$ is a reason for φ_μ unsatisfiability.*
3. *If $l \in \nu$, l is existential and monotone in φ_μ, then $\nu \setminus \{l\}$ is a reason for φ_μ unsatisfiability.*
4. *If $l \in \nu$, $\overline{l} \in \nu'$, ν' is a reason for $\varphi_{\mu;\overline{l}}$ unsatisfiability, and l is existential, then $(\nu \cup \nu') \setminus \{l, \overline{l}\}$ is a reason for φ_μ unsatisfiability.*

If φ is (2), considering Theorem 2 and Figure 2:

- given what we said in the paragraph below Theorem 1, the 4th statement allows us to conclude that $\{x_1, \neg x_3\}$ is a reason for φ_μ unsatisfiability when μ is $x_1; \neg x_2; x_5; \neg x_3$,
- from the above, the 2nd statement allows us to conclude that $\{x_1\}$ is a reason for φ_μ unsatisfiability when μ is $x_1; \neg x_2; x_5$,
- from the above, the 1st statement allows us to conclude that $\{x_1\}$ is a reason for φ_μ unsatisfiability when μ is in $\{x_1; \neg x_2, x_1\}$.

From the last item, it follows that looking for assignments satisfying φ_μ when μ begins with x_1, is useless. Given this, our "backjumping" procedure would have avoided the generation of the branch leading to ($F8$) in Figure 2.

4.2 Solution-directed Backjumping

The following Theorem allows us to compute the reason for φ_μ result when the matrix of φ_μ is empty. Given a sequence of literals μ, we say that a literal l is TRUE-*irrelevant* in μ if —assuming μ' is the sequence obtained by deleting l in μ— the matrix of $\varphi_{\mu'}$ is empty.

Theorem 3 *Let φ be a QBF. Let μ be an assignment for φ such that the matrix of φ_μ is empty. Let μ' be a sequence obtained by recursively eliminating a* TRUE*-irrelevant literal in μ. Let ν be the set of literals in μ'. Then ν is a reason for φ_μ satisfiability.*

With reference to Figure 2, the above Theorem allows us to conclude that, e.g., $\{\neg x_1, x_4, x_5\}$ is a reason for φ_μ satisfiability, if μ is $\neg x_1; \neg x_3; x_4; x_5$ and φ is (2).

Theorem 4 *Let φ be a QBF. Let l be a literal. Let $\mu; l$ be an assignment for φ. Let ν be a reason for $\varphi_{\mu;l}$ satisfiability.*

1. *If l is not in ν, then ν is a reason for φ_μ satisfiability.*
2. *If $l \in \nu$, and l is existential, then $\nu \setminus \{l\}$ is a reason for φ_μ satisfiability.*
3. *If $l \in \nu$, l is universal and monotone in φ_μ, then $\nu \setminus \{l\}$ is a reason for φ_μ satisfiability.*
4. *If $l \in \nu$, $\overline{l} \in \nu'$, ν' is a reason for $\varphi_{\mu;\overline{l}}$ satisfiability, and l is universal, then $(\nu \cup \nu') \setminus \{l, \overline{l}\}$ is a reason for φ_μ satisfiability.*

If φ is (2), considering Theorem 4 and Figure 2:

```
1  function Backjump(res)
2    wr := InitWr(res);
3    while (⟨Stack is not empty⟩)
4      l := Retract();
5      if (l ∈ wr)
6        if (res = FALSE and |l|.type = ∃) or
7            (res = TRUE and |l|.type = ∀)
8          if (|l|.mode = UNIT) or (|l|.mode = R-SPLIT)
9            wr := (wr ∪ |l|.reason) \ {l, l̄}
10         if(|l|.mode = L-SPLIT)
11           |l|.mode := R-SPLIT;
12           |l|.reason := wr;
13           return l̄;
14       else wr := wr \ {l};
15   return NULL;
```

Figure 3: A procedure for conflict-directed and solution-directed backjumping.

- given what we said in the paragraph below Theorem 3, the 2nd statement allows us to conclude that $\{\neg x_1, x_4\}$ is a reason for φ_μ satisfiability when μ is $\neg x_1; \neg x_3; x_4$,
- from the above, the 2nd statement allows us to conclude that $\{\neg x_1\}$ is a reason for φ_μ satisfiability when μ is $\neg x_1; \neg x_3$,
- from the above, the 1st statement allows us to conclude that $\{\neg x_1\}$ is a reason for φ_μ satisfiability when μ is $\neg x_1$.

From the last item, it follows that looking for assignments falsifying φ_μ when μ begins with $\neg x_1$, is useless. Given this, our "backjumping" procedure would have avoided the generation of the branch leading to $(F5)$ in Figure 2.

5 Implementation in QuBE

A procedure incorporating both conflict-directed and solution-directed backjumping has been implemented in QuBE. A high-level description of this procedure is presented in Figure 3. Consider Figure 3. Assume that φ is the input QBF and that $\mu = l_1; \ldots; l_m$ is the assignment for φ corresponding to the sequence of literals stored in the stack. Then,

- *InitWr*(*res*) initializes the variable wr, storing the reason for φ_μ result. In our implementation:
  - if *res* = FALSE, φ_μ contains a contradictory clause. Let C be a clause in φ such that, for each literal l in C, $\overline{l}$ is in μ or l is universal in φ. Then *InitWr*(*res*) returns the set of literals l in μ such that $\overline{l}$ is in C (see Theorem 1). For example, if φ is (2), and μ is $x_1; \neg x_2; x_5; \neg x_3; x_4$, then *InitWr*(*res*) returns $\{x_1, \neg x_3, x_4\}$.
  - if *res* = TRUE, the matrix of φ_μ is empty. Then *InitWr*(*res*) returns the set of literals in the sequence obtained from μ by recursively eliminating a universal literal l such that, for each clause C in φ, if $l \in C$ then there is another literal l' in the sequence with $l' \in C$ (see Theorem 3). For example, if φ is (2), and μ is $\neg x_1; \neg x_3; x_4; x_5$, then *InitWr*(*res*) returns $\{\neg x_1, x_4, x_5\}$.
- If l is a literal l_i in μ, $|l|.reason$ is the property of $|l|$ that, if set, stores the reason for $\varphi_{l_1;\ldots;l_i}$ result.

The function *QubeSolver* in Figure 1 needs to be modified in two ways. First, if l is a unit in φ_μ, then $\varphi_{\mu;\overline{l}}$ contains a contradictory clause. Let C be a clause in φ such that, for each literal l' in C, $\overline{l'}$ is in $\mu; \overline{l}$ or l' is universal in φ. Then, the set ν of literals l' in $\mu; \overline{l}$ such that $\overline{l'}$ is in C is a reason for $\varphi_{\mu;\overline{l}}$ unsatisfiability. If *UnitSetReason*(l) is invoked when l is a unit in φ_μ, and assuming this function returns a set ν defined as above, the instruction

$$|l|.reason := \textit{UnitSetReason}(l);$$

has to be added in line 8. For example, if φ is (2), and μ is $x_1; \neg x_2; x_5; \neg x_3$, then x_4 is a unit in φ_μ, $\varphi_{\mu;\neg x_4}$ contains a contradictory clause, and $\{x_1, \neg x_3, \neg x_4\}$ is stored in $x_4.reason$. Second, the procedure *Backjump*(*res*) has to be invoked in place of *Backtrack*(*res*) at line 25.

Considering the procedure *Backjump*(*res*) in Figure 3 — once the working reason is initialized (line 2) according to the above— *Backjump*(*res*) pops all the literals and corresponding QBFs (line 4) from the stack, till a literal l is reached such that l belongs to the working reason wr (line 5), $|l|.mode$ is L-SPLIT (line 10), and either

- l is existential and *res*=FALSE (line 6); or
- l is universal and *res*=TRUE (line 7).

If such a literal l exists, $|l|.mode$ is set to R-SPLIT(line 11), the working reason is stored in $|l|.reason$ (line 12), and $\overline{l}$ is returned (line 13). If no such literal exists, NULL is returned (line 15).

Notice that if l is not in wr, we can safely retract l despite the other conditions (see statement 1 in Theorems 2, 4): Assigning $\overline{l}$ would not change the result of the computation.

If l is in wr, one of the conditions in line 6 or line 7 is satisfied, but $|l|.mode$ is UNIT or R-SPLIT (line 8), we can use the results in statement 4 of Theorems 2, 4 to compute the new working reason (line 9).

If l is in wr but neither the condition in line 6 nor the condition in line 7 is satisfied, then we can retract l and remove l from wr (line 14). See the second statement of Theorems 2, 4.

Finally, given the reasons returned by our implementation of *InitWr*(*res*), it is easy to see that neither a universal, monotone literal can belong to the reason of an universal literal when *res*=TRUE, nor an existential, monotone literal can belong to the reason of an existential literal when *res*=FALSE. Thus, the statement 3 in Theorems 2, 4 (included for theoretical completeness) is never applied in our implementation.

Our implementation of QuBE with *Backjump* is a generalization of the conflict-directed backjumping procedure implemented, e.g., in RELSAT: assuming that our procedure and RELSAT –without learning– perform the same nondeterministic choices, the two systems have the same behavior on QBFs without universal quantifiers.

6 Experimental analysis

To evaluate the benefits deriving from backjumping, we compare QuBE with backtracking (that we call QuBE-BT), and QuBE with backjumping (that we call QuBE-BJ). For both

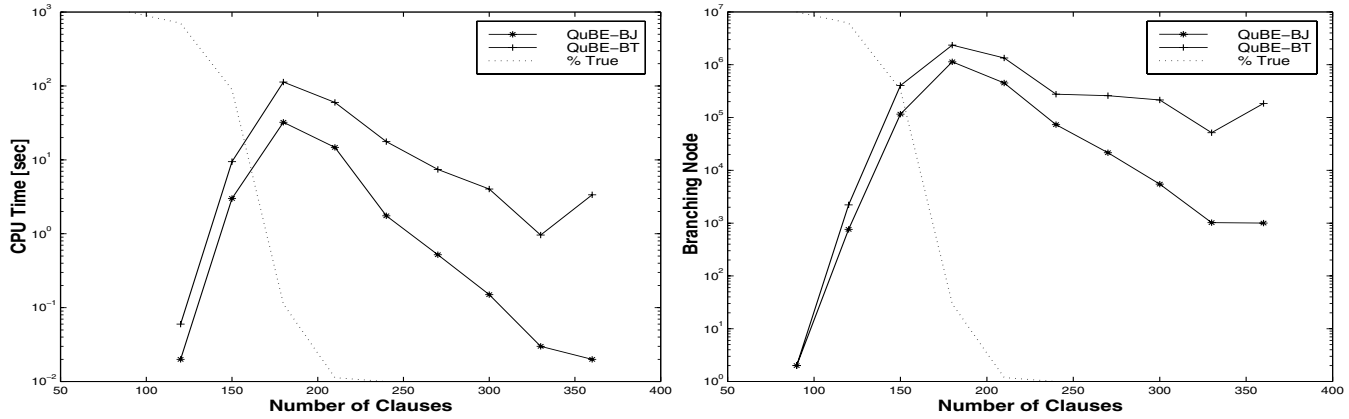

Figure 4: QUBE-BT and QUBE-BJ median CPU time (left) and number of branching nodes (right). 100 samples/point. Background: satisfiability percentage.

systems, we use the branching heuristics described in [Feldmann *et al.*, 2000]). All the tests have been run on a Pentium III, 600MHz, 128MBRAM.

We first consider sets of randomly generated QBFs. The generation model that we use is model A by Gent and Walsh [1999]. In this model, each QBF has the following 4 properties:

1. the prefix consists of k sequences, each sequence has n quantifiers, and each two quantifiers in a same sequence, are of the same type,
2. the rightmost quantifier is $\exists$,
3. the matrix consists of l clauses,
4. each clause consists of h literals of which at least 2 are existential.

Figure 4 shows the median of the CPU times (left) and number of branching nodes (right) of QUBE-BT and QUBE-BJ when $k = 4$, $n = 30$, $h = 5$, and l (on the x-axis) is varied in such a way to empirically cover the "100% satisfiable – 100% unsatisfiable" transition (shown in the background). Notice the logarithmic scale on the y-axis. Consider Figure 4-left. As it can be observed, QUBE-BJ is faster (up-to two orders of magnitude) than QUBE-BT. QUBE-BJ better performances are due to its minor number of branching nodes, as shown in Figure 4-right.

	QUBE-BT		QUBE-BJ	
Test File	Time	Br. Nodes	Time	Br. Nodes
B*-3ii.4.3	>1200	–	2.340	59390
B*-3ii.5.2	>1200	–	53.500	525490
B*-3iii.4	>1200	–	0.560	18952
T*-6.1.iv.11	31.990	500471	6.860	139505
T*-6.1.iv.12	22.700	298172	2.660	56387
T*-7.1.iv.13	755.760	8932409	117.380	1948541
T*-7.1.iv.14	466.600	4729695	38.720	688621
C*-23.24	604.020	8388653	706.430	8388653

Table 1: QUBE-BT and QUBE-BJ on Rintanen's benchmarks. Names have been abbreviated to fit in the table.

We also test QUBE-BT and QUBE-BJ on the structured Rintanen's benchmarks.[3] These problems are translations from planning problems to QBF (see [Rintanen, 1999a]). For lack of space, Table 1 shows QUBE-BT and QUBE-BJ performances on only 8 out of the 38 benchmarks available. On 13 of the other 30 problems, QUBE-BT and QUBE-BJ perform the same number of branching nodes, and on the remaining 17, none of the two systems is able to solve the problem in the time limit of 1200 seconds. As before, QUBE-BJ never performs more branching nodes than QUBE-BT, and is sometimes much faster. The last line of the table shows that when the number of branching nodes performed by QUBE-BT and QUBE-BJ is the same, the computational overhead paid by QUBE-BJ is not dramatic.

7 Conclusions and Future work

In the paper we have shown that it is possible to generalize the conflict-directed backjumping schema for SAT to QBFs. We have introduced solution-directed backjumping. Our implementation in QUBE shows that these forms of backjumping can produce significant speed ups. As far as we know, this is the first time a backjumping schema has been proposed, implemented and experimented for QBFs. It is worth remarking that the logics of conflict-directed (section 4.1) and solution-directed (section 4.2) backjumping are symmetrical: Indeed, it would have been easy to provide a uniform treatment accounting for both forms of backjumping. We decided not to do it in order to improve the readability of the paper, at the same time showing the tight relations with the conflict-directed backjumping schema for SAT.

Beside the above results, we have also comparatively tested QUBE and some of the other QBF solvers mentioned in the introduction. Our results show that QUBE compares well with respect to the other deciders, even without backjumping. For example, of the 38 Rintanen's structured problems, DECIDE, QUBE-BJ, QUBE-BT, QSOLVE, EVALUATE are able to solve 35, 21, 18, 11, 0 samples respectively in less than 1200s. In this regard, we point out that DECIDE features

[3] Available at `www.informatik.uni-freiburg.de/~rintanen/qbf.html`.

"inversion of quantifiers" and "sampling" mechanisms which seem particularly effective on these benchmarks.

QuBE, the experimental results, and the test sets used are available at QuBE web page:

`www.mrg.dist.unige.it/star/qube`.

QuBE, besides the backjumping procedure above described, features six different branching heuristics, plus an adaptation of trivial truth (see [Cadoli *et al.*, 1998, Cadoli *et al.*, 2000]). (See [Giunchiglia *et al.*, 2001b] for a description of QuBE's available options.) About trivial truth, backjumping and their interactions, we have conducted an experimental evaluation on this. The result is that neither trivial truth is always better than backjumping, nor the other way around. On the other hand, the overhead of each of these techniques (on the test sets we have tried) is not dramatic, and thus it seems a good idea to use both of them. These and other results are reported in [Giunchiglia *et al.*, 2001c].

About the future work, we are currently investigating the effects of using different branching heuristics. Then, we plan to extend QuBE in order to do some form of "size" or "relevance" learning as it has been done in SAT (see, e.g., [Bayardo, Jr. and Schrag, 1997]).

Acknowledgments

Thanks to Davide Zambonin for the many fruitful discussion about implementation. Thanks to an anonymous reviewer for suggesting the name "solution-directed backjumping". Thanks to Marco Cadoli, Rainer Feldmann, Theodor Lettman, Jussi Rintanen, Marco Schaerf and Stefan Schamberger for providing us with their systems and helping us to figure them out during our experimental analisys. This work has been partially supported by ASI and MURST.

References

[Bayardo, Jr. and Schrag, 1997] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. AAAI*, pages 203–208, 1997.

[Cadoli *et al.*, 1998] M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proc. AAAI*, 1998.

[Cadoli *et al.*, 2000] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An algorithm to evaluate quantified boolean formulae and its experimental evaluation. *Journal of Automated Reasoning*, 2000. To appear. Reprinted in [Gent *et al.*, 2000].

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[Dechter, 1990] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990.

[Egly *et al.*, 2000] U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving advanced reasoning tasks using quantified boolean formulas. In *Proc. AAAI*, 2000.

[Feldmann *et al.*, 2000] R. Feldmann, B. Monien, and S. Schamberger. A distributed algorithm to evaluate quantified boolean formulae. In *Proc. AAAI*, 2000.

[Gent and Walsh, 1999] Ian Gent and Toby Walsh. Beyond NP: the QSAT phase transition. In *Proc. AAAI*, pages 648–653, 1999.

[Gent *et al.*, 2000] Ian P. Gent, Hans Van Maaren, and Toby Walsh, editors. *SAT2000. Highlights of Satisfiability Research in the Year 2000*. IOS Press, 2000.

[Giunchiglia *et al.*, 2001a] Enrico Giunchiglia, Marco Maratea, Armando Tacchella, and Davide Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proc. of the International Joint Conference on Automated Reasoning (IJCAR'2001)*, 2001.

[Giunchiglia *et al.*, 2001b] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QuBE: A system for deciding quantified boolean formulas satisfiability. In *Proc. of the International Joint Conference on Automated Reasoning (IJCAR'2001)*, 2001.

[Giunchiglia *et al.*, 2001c] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. On the effectiveness of backjumping and trivial truth in quantified boolean formulas satisfiability, 2001. Submitted. Available at `www.mrg.dist.unige.it/star/qube`.

[Kleine-Büning, H. and Karpinski, M. and Flögel, A., 1995] Kleine-Büning, H. and Karpinski, M. and Flögel, A. Resolution for quantified boolean formulas. *Information and computation*, 117(1):12–18, 1995.

[Prosser, 1993] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[Rintanen, 1999a] Jussi Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[Rintanen, 1999b] Jussi T. Rintanen. Improvements to the evaluation of quantified boolean formulae. In *Proc. IJCAI*, pages 1192–1197, 1999.

Solving Non-Boolean Satisfiability Problems with Stochastic Local Search

Alan M. Frisch and Timothy J. Peugniez
Artificial Intelligence Group
Department of Computer Science
University of York
York YO10 5DD
United Kingdom

Abstract

Much excitement has been generated by the recent success of stochastic local search procedures at finding satisfying assignments to large formulas. Many of the problems on which these methods have been effective are non-Boolean in that they are most naturally formulated in terms of variables with domain sizes greater than two. To tackle such a problem with a Boolean procedure the problem is first reformulated as an equivalent Boolean problem. This paper introduces and studies the alternative of extending a Boolean stochastic local search procedure to operate directly on non-Boolean problems. It then compares the non-Boolean representation to three Boolean representations and presents experimental evidence that the non-Boolean method is often superior for problems with large domain sizes.

1 Introduction

Much excitement has been generated by the recent success of stochastic local search (SLS) procedures at finding satisfying truth assignments to large formulas of propositional logic. These procedures stochasticly search a space of all assignments for one that satisfies the given formula. Many of the problems on which these methods have been effective are non-Boolean in that they are most naturally formulated in terms of variables with domain sizes greater than two. To tackle a non-Boolean problem with a Boolean procedure, the problem is first reformulated as an equivalent Boolean problem in which multiple Boolean variables are used in place of each non-Boolean variable.

This encode-and-solve approach often results in comparable, if not superior, performance to solving the problem directly. Because Boolean satisfiability is conceptually simple, algorithms for it are often easier to design, implement and evaluate. And because SLS algorithms for Boolean satisfiability have been studied intensively for more than a decade, highly-optimised implementations are publicly available.

This paper proposes and studies a new approach to solving non-Boolean satisfaction problems: that of generalising a Boolean SLS procedure to operate directly on a non-Boolean formula by searching through a space of assignments to non-Boolean variables. In particular, we have generalised Walksat [Selman *et al.*, 1994], a highly-successful SLS procedure for Boolean satisfiability problems, to a new procedure, NB-Walksat [Peugniez, 1998; Frisch and Peugniez, 1998], that works on formulas whose variables have domains of any finite size.[1]

In this way we are able to apply highly-refined SLS technology directly to non-Boolean problems without having to encode non-Boolean variables as Boolean variables.

The main question addressed by this paper is how the performance of the direct approach compares to that of the transformational (or encode and solve) approach. In particular we compare one direct method, NB-Walksat, and three transformational methods by empirically testing their ability to solve large graph colouring problems and large random non-Boolean formulas. Our three transformation methods consist of applying Walksat to the results of three transforms.

Boolean variables are merely a special case of non-Boolean variables, and, intuitively, the difference between the non-Boolean and Boolean variables grows as the domain size of the non-Boolean variable increases. Consequently, one would expect that in a comparison of encodings for non-Boolean problems that domain size would be the most important parameter to consider and that one would find that any difference in performance between the encodings would increase when domain size is increased. Ours is the first study to consider this.

Our experimental results show NB-Walksat to be highly effective, demonstrating that the effectiveness of the Walksat strategies can be transferred from the Boolean case to the non-Boolean case. On problems with large domain sizes our direct method is often superior to the transformation methods, which, in some cases, are ineffective.

In the course of studying this new method of solving non-Boolean problems with SLS, we make several other new contributions. Most naturally, we have had to generalise the notion of a Boolean formula to that of a non-Boolean formula. Of the three transformations we use, one is new and one is an enhanced version of a transformation used by others. In order to test the effect of domain size on problem solving performance we want a method for generating random formulas that vary in domain size but are similar in other respects. We

[1] NB-Walksat and a suite of supporting programs are available at http://www.cs.york.ac.uk/~frisch/NB.

propose such a method and use it in our experiments.

2 Non-Boolean Formulas

Non-Boolean formulas are constructed from propositional variables, each of which is associated with a finite, non-empty domain. A non-Boolean assignment maps every variable to a member of its domain. Atomic non-Boolean formulas (or nb-atoms) are of the form X/d, where X is a variable and d is a member of its domain. This formula is assigned *TRUE* by an assignment if the assignment maps X to d; otherwise it is assigned *FALSE*. Molecular non-Boolean formulas are constructed from atomic non-Boolean formulas and truth-functional connectives with the usual syntax and Boolean semantics. Observe that though non-Boolean variables can take on values from a domain of arbitrary size, the semantics is still two-valued in that every formula is assigned either *TRUE* or *FALSE*. As usual, we say that an assignment satisfies a non-Boolean formula if it maps it to *TRUE*.

Walksat, and many other Boolean SLS procedures, operate on Boolean formulas in conjunctive normal form (CNF), and NB-Walksat, our generalisation of Walksat, operates on non-Boolean formulas in CNF. A formula, Boolean or non-Boolean, is in CNF if it is a conjunction of disjunctions of literals. A literal is either an atomic formula (called a positive literal) or its negation (called a negative literal). We say that a CNF formula is *positive* if all its literals are positive and *negative* if all its literals are negative.

Non-Boolean formulas generalise Boolean formulas since a Boolean formula can be transformed to a non-Boolean formula by replacing every atom P with $P'/$*TRUE*, where P' is a variable whose domain is $\{$*TRUE*, *FALSE*$\}$.

We sometimes use terms such as "nb-atom" or "nb-formula" to emphasise that these syntactic objects are part of the non-Boolean language. Similar use is made of terms such as "b-atom" and "b-formula".

3 NB-Walksat

Walksat is a highly successful SLS procedure for finding satisfying assignments to Boolean formulas in CNF. We have generalised Walksat to a new system, NB-Walksat, that operates similarly on non-Boolean formulas. Indeed when handling a Boolean formula the two procedures perform the same search.[2] This paper is concerned with version 4 of NB-Walksat, which we derived from Walksat version 19. This section describes the operation of NB-Walksat and, since on Boolean formulas NB-Walksat and Walksat perform the same search, this section implicitly describes the operation of Walksat.

The simplest way to understand the operation of NB-Walksat is to consider it as working on positive CNF nb-formulas. This can be achieved by considering NB-Walksat's first step to be the replacement of every negative literal $\neg X/d_i$ with $X/d_1 \vee \cdots \vee X/d_{i-1} \vee X/d_{i+1} \vee \cdots \vee X/d_n$, where X is a variable with domain $\{d_1, \ldots, d_n\}$.

[2] We used this property to help test that NB-Walksat was correctly implemented.

NB-Walksat operates by choosing a random assignment and then, until a satisfying assignment is found, repeatedly selecting a literal from an unsatisfied clause and modifying the assignment so as to satisfy that literal. Since the selected literal, X/d, occurs in an unsatisfied clause, the present assignment must map X to a value other than d. The present assignment is modified so that it maps X to d, and its mapping of all other variable is unmodified. We say that the literal X/d has been *flipped*.

What distinguishes NB-Walksat and Walksat from other procedures is the heuristic employed for selecting which literal to flip. Though Walksat provides a range of such heuristics, the greedy heuristic is generally the most effective [Selman *et al.*, 1994] and many reported experiments have used it. Consequently, it is the greedy version of Walksat that forms the basis for NB-Walksat and only the greedy heuristic is considered in this paper.

NB-Walksat with the greedy heuristic chooses a literal to flip by first randomly selecting a clause with uniform distribution from among all the clauses that are not satisfied by the current assignment. We say that flipping a literal *breaks* a clause if the clause is satisfied by the assignment before the flip but not after the flip. If the selected clause contains a literal such that flipping it would break no clauses, then the literal to flip is chosen randomly with uniform distribution from among all such literals. If the selected clause contains no such literals, then a literal is chosen either (i) randomly with uniform distribution from the set of all literals in the clause or (ii) randomly with uniform distribution from among the literals in that clause that if flipped would break the fewest clauses. The decision to do (i) or (ii) is made randomly; with a user-specified probability, P_{noise}, the "noisy" choice (i) is taken.

4 Transforming Non-Boolean Formulas

To transform nb-satisfaction problems to b-satisfaction problems we map each nb-formula to a b-formula such that the satisfying assignments of the two formulas correspond. This paper presents three such transforms, called the *unary/unary, unary/binary* and *binary* transforms. Each operates on an arbitrary formula, though our experiments only apply the transforms to CNF formulas. Each transform operates by replacing each atom in the nb-formula with a b-formula that, in a sense, encodes the nb-atom it replaces. The resulting formula is known as the *kernel* of the transformation. The transforms employ two ways of producing a kernel; we call these two encodings "unary" and "binary".

If the unary encoding of the kernel is used, the transform also needs to conjoin two additional formulas to the kernel, known as the *at-least-one* formula (or ALO formula) and the *at-most-one* formula (or AMO formula). As with the kernel, two encodings can be used for the ALO and AMO formulas: unary and binary. The three transforms we use in this paper are *binary* (which uses a binary encoding for the kernel and no ALO or AMO formula), *unary/unary* (which uses unary encodings for the kernel and for the ALO and AMO formulas) and *unary/binary* (which uses a unary encoding for the kernel and a binary encoding for the ALO and AMO formulas).

The longer version of this paper analyses the sizes of the formulas produced by all the encodings.

The Unary/Unary Transform The unary/unary transform produces a kernel by transforming each nb-atom X/d to a distinct propositional variable, which we shall call $X{:}d$. The idea is that a Boolean assignment maps $X{:}d$ to *TRUE* if and only if the corresponding non-Boolean assignment maps X to d. Thus, the role of an nb-variable with domain $\{d_1, \ldots d_n\}$ is played by n b-variables.

Furthermore, one must generally add additional formulas to the Boolean encoding to represent the constraint that a satisfying assignment must satisfy exactly one of $X{:}d_1, \ldots, X{:}d_n$. This constraint is expressed as a conjunction of an ALO formula (asserting that at least one of the variables is true) and an AMO formula (asserting that at most one of the variables is true). To state that at *least* one of $X{:}d_1, \ldots, X{:}d_n$ must be satisfied we simply use the clause $X{:}d_1 \vee \cdots \vee X{:}d_n$. The entire ALO formula is a conjunction of such clauses, one clause for each nb-variable. To say that at *most* one of $X{:}d_1, \ldots, X{:}d_n$ must be satisfied we add $\neg X{:}d_i \vee \neg X{:}d_j$, for all i and j such that $1 \leq i < j \leq n$. The entire AMO formula is a conjunction of all such clauses produced by all nb-variables.

Notice that these ALO and AMO formulas are in CNF. And since the transform produces a kernel whose form is identical to that of the original formula, the entire b-formula produced by the unary/unary transform is in CNF if and only if the original nb-formula is.

The Binary Transform The unary/unary transform uses D b-variables to encode a single nb-variable of domain size D and, hence, uses a base 1 encoding. By using a base 2 encoding, the binary transformation requires only $\lceil \log_2 D \rceil$ b-variables to encode the same nb-variable.[3] If X is a variable of domain size D then the binary transform maps an nb-literal of the form X/d_i by taking the binary representation of $i-1$ and encoding this in $\lceil \log_2 D \rceil$ Boolean variables. For example, if X has domain $\{d_1, d_2, d_3, d_4\}$ then

X/d_1 is mapped to $\neg X_2 \wedge \neg X_1$,
X/d_2 is mapped to $\neg X_2 \wedge X_1$,
X/d_3 is mapped to $X_2 \wedge \neg X_1$ and
X/d_4 is mapped to $X_2 \wedge X_1$.

To see what happens when the domain size is not a power of two, reconsider X to have the domain $\{d_1, d_2, d_3\}$. If we map X/d_1, X/d_2 and X/d_3 as above then there is a problem in that the Boolean assignment that satisfies $X_2 \wedge X_1$ does not correspond to an assignment of a domain value to X. One solution to this problem, which has been employed by Hoos [1998], is to add an ALO formula to ensure that the *extraneous* binary combination $X_2 \wedge X_1$ cannot be satisfied in any solution.

Here we introduce and use a new and better solution to this problem in which no extraneous combinations are produced and therefore no ALO formula is required. In this example the extraneous combination is eliminated if

X/d_1 is mapped to $\neg X_2$,
X/d_2 is mapped to $X_2 \wedge \neg X_1$ and
X/d_3 is mapped to $X_2 \wedge X_1$.

Here, the transform of X/d_1 covers two binary combinations: $(\neg X_2 \wedge X_1)$ and $(\neg X_2 \wedge \neg X_1)$.

To see what happens in general let X be a variable with domain $\{d_1, \ldots, d_n\}$ and let k be $2^{\lceil \log_2 n \rceil} - n$. Then $X/d_1, \ldots, X/d_k$ are each mapped to cover two binary combinations and $X/d_{k+1}, \ldots, X/d_n$ are each mapped to cover a single binary combination.

Notice that this transform generates no extraneous binary combinations. Also notice that, as a special case, if n is a power of two then each X/d_i $(1 \leq i \leq n)$ is mapped to cover a single binary combination. Finally, to confirm that this binary transform requires no AMO formula and no ALO formula, observe that every Boolean assignment must satisfy the binary transform of exactly one of $X/d_1, \ldots, X/d_n$.

Notice that the binary transformation of a CNF formula is not necessarily in CNF. However, the binary transformation of a negative CNF formula is *almost* in CNF; it is a conjunction of disjunctions of negated conjunctions of literals. By using DeMorgan's law, the negations can be moved inside of the innermost conjunctions, resulting in a CNF formula of the same size. At the other extreme, the binary transformation of a positive CNF formula is a conjunction of disjunctions of conjunctions of literals. One way of transforming this to CNF is to distribute the disjunctions over the conjunctions. Unfortunately applying this distribution process to the binary transform of a positive nb-clause produces a CNF formula that can be exponentially larger than the original clause.

The Unary/Binary Transform The unary/binary transform produces the same kernel as the unary/unary transform. The ALO and AMO formulas it produces achieve their effect by introducing the binary encodings of nb-atoms and linking each to the unary encoding of the same nb-atom. Since the binary encoding requires no ALO or AMO formulas, the unary/binary encoding requires no ALO or AMO formulas beyond these linking formulas.

The links provided by the AMO formula state that for each nb-atom, X/d_i, its unary encoding, $X{:}d_i$, implies its binary encoding. For example, if the nb-variable X has domain $\{d_1, d_2, d_3, d_4\}$ then the AMO linking formula for X is

$$(X{:}d_1 \rightarrow (\neg X_2 \wedge \neg X_1)) \quad \wedge \quad (X{:}d_2 \rightarrow (\neg X_2 \wedge X_1)) \quad \wedge$$
$$(X{:}d_3 \rightarrow (X_2 \wedge \neg X_1)) \quad \wedge \quad (X{:}d_4 \rightarrow (X_2 \wedge X_1)).$$

The entire AMO formula is a conjunction of such linking formulas, one for each nb-variable.

The links provided by the ALO formula state that for each nb-atom, X/d_i, its unary encoding $X{:}d_i$ is implied by its binary encoding. For example, if the nb-variable X has domain $\{d_1, d_2, d_3, d_4\}$ then the ALO-linking formula for X is

$$((\neg X_2 \wedge \neg X_1) \rightarrow X{:}d_1) \quad \wedge \quad ((\neg X_2 \wedge X_1) \rightarrow X{:}d_2) \quad \wedge$$
$$((X_2 \wedge \neg X_1) \rightarrow X{:}d_3) \quad \wedge \quad ((X_2 \wedge X_1) \rightarrow X{:}d_4).$$

The entire ALO formula is a conjunction of linking formulas, one linking formula for each nb-variable. Observe that both

[3] We write $\lceil x \rceil$ to denote the smallest integer that is greater than or equal to x.

the AMO and ALO linking formulas can be put easily into CNF with at most linear expansion.

5 Experimental Results

This section presents experiments on graph colouring problems and random nb-formulas that compare the performance of the four methods, which we shall refer to as NB (non-Boolean encoding), BB (binary encoding) UU (unary/unary encoding) and UB (unary/binary encoding). In all experiments, Walksat version 35 was used to solve the Boolean encodings and NB-Walksat version 4 was used to solve the non-Boolean encodings, even in cases where the domain size is 2. Both programs provide the user the option of either compiling the program with fixed size data structures or allocating the data structures when the formula is input at runtime; the latter option was used in all experiments. All experiments were run on a 600 MHz Pentium III with 126 megabytes of memory.

Considerable care must be taken in setting the P_{noise} parameter for the experiments. Much work in this area has been reported without giving the value used for P_{noise}, and thus is irreproducible. Setting the parameter to any fixed value over all formulas is not acceptable; we have observed that a parameter setting that is optimal for one formula can be sub-optimal by several orders of magnitude for another formula. The best option is to report performance at the optimal setting for P_{noise}, which—in the absence of any known method to determine this *a priori*—we have determined experimentally. This is also the route followed by Hoos [1998] in his extremely careful work.

In using SLS procedures it is often considered advantageous to restart the search at a new, randomly-selected assignment if the procedure has not found a solution after a prescribed number of flips. This issue is not relevant for the current study. Since the optimal restart point and the amount gained by restarting are functions of the procedure's performance without restarts, this study need only be concerned with the performance without restarts.

Graph Colouring Our experiments used 6 instances of the graph colouring problem that are publicly available at the Rutgers Univ. DIMACS FTP site.[4]

Each problem instance was encoded as a CNF nb-formula. For each node in the graph the formula uses a distinct variable whose domain is the set of colours. The formula itself is a conjunction of all clauses of the form $\neg X/c \vee \neg Y/c$, such that X and Y are nodes connected by an arc and c is a colour.

Three Boolean CNF representations of each problem instance were produced by applying the three transforms of Section 4 to the non-Boolean representation. The longer version of this paper proves that a negative CNF nb-formula can be transformed to a b-formula without the AMO formulas. So the two unary encodings used here contain only kernels and ALO clauses. Also because the nb-clauses representing instances of the graph colouring problem are negative, the binary transform maps each nb-clause to a single b-clause (as discussed in Section 4).

[4] ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color/

Figure 1 shows the results obtained for solving each problem instance 101 times with each of the four methods. The formula size column shows the size (in terms of atom occurrences) of the formula input to Walksat or NB-Walksat. All experiments were run with P_{noise} set to its optimal value, which is recored in the P_{noise} column. The flip rate column gives the mean flip rate over all 101 runs. The median flips and median time column gives the sample median of the number of flips and amount of cpu time taken to solve the problem.

Examination of these results reveals that over the range of problems NB and UU have roughly comparable (within a factor of 2 to 3) solution times.[5] With increasing domain size (number of colours) the NB fliprate drops sharply but the advantage that NB has in terms of number of flips increases. These two opposing factors cancel each other, resulting in the roughly comparable solution times.

On the two problems with small domain size (5 colours) BB and UB produce solution times that are competitive with the other methods. However, on three problems with larger domain size—le450_15a with 15 colours and DSJC125.5.col with 17 and 18 colours—BB and UB do not compete well with the other methods. On le450_25a, a very easy problem with a domain size of 25, BB has competitive performance but UB is totally ineffective. We conjecture that UB cannot solve this very easy problem because the method is highly sensitive to domain size.

Random Non-Boolean CNF Formulas Since we can control certain parameters in the generation of random CNF nb-formulas, they provide a good testbed. In particular, since this paper is a study of solving problems with domain sizes greater than two, we would like to know how problem solving performance varies with domain size. To measure this we need to select problem instances that have different domain sizes but are otherwise similar. Formulating a notion of "otherwise similar" has been one of the most challenging problems faced by this research.

We have developed a program that generates random, positive CNF nb-formulas using five parameters: N, D, C, V and L. Each generated formula consists of exactly C clauses. Using a fixed set of N variables each with a domain size of D, each clause is generated by randomly choosing V distinct variables and then, for each variable, randomly choosing L distinct values from its domain. Each of the chosen variables is coupled with each of its L chosen values to form $L \cdot V$ positive literals, which are disjoined together to form a clause.

The simplest conjecture on how to study varying domain size is to fix the values of N, C, V, and L and then to systematically vary D. One can easily see that performance on this task would just exhibit typical phase-transition behaviour [Mitchell *et al.*, 1992]: small values of D would produce underconstrained problems, which would become critically-constrained and then over-constrained as D increases. The

[5] We caution the reader against taking solution time too seriously as we have found that the ratio of solution times between computers varies greatly from formula to formula.

Problem instance	method	P_{noise} setting	formula size	flip rate (flips/sec)	median flips	median time (secs)
le450_5a	NB	.560	57,140	34,000	242,000	7.12
5 colours	BB	.465	137,136	51,000	309,000	6.06
450 nodes	UU	.600	59,390	119,000	350,000	2.95
5714 arcs	UB	.453	64,790	232,000	1,700,000	7.34
le450_5d	NB	.703	97,570	17,700	98,700	5.58
5 colours	BB	.605	234,168	26,100	68,400	2.62
450 nodes	UU	.719	99,820	65,500	155,000	2.37
9757 arcs	UB	.597	105,220	132,000	729,000	5.52
le450_15a	NB	.088	245,040	15,400	196,000	12.7
15 colours	BB		963,824			
450 nodes	UU	.131	251,790	53,900	1,268,000	23.5
8168 arcs	UB		278,340			
DSJC125.5.col	NB	.177	132,294	9,490	495,000	52.2
17 colours	BB		544,740			
125 nodes	UU	.11	134,419	49,000	3,350,000	68.4
3891 arcs	UB		143,169			
DSJC125.5.col	NB	.202	140,076	8,920	6,850	0.768
18 colours	BB	.050	591,432	15,200	1,160,000	76.1
125 nodes	UU	.140	142,326	46,200	34,700	0.751
3891 arcs	UB	.016	151,826	273,000	1,590,000	5.82
le450_25a	NB	.120	413,000	2,840	638	.224
25 colours	BB	.088	1,949,360	6,460	2,900	0.449
450 nodes	UU	.163	424,250	27,400	11,700	0.427
8260 arcs	UB		477,350			

Figure 1: Results, given to three significant figures, for 101 sample runs on each graph colouring problem. On those rows missing entries, 11 attempts, each of 10,000,000 flips, were made at each of four P_{noise} settings (.083, .167, .25 and .33); no solutions were found.

problem instances generated by this method would not be similar in terms of their location relative the phase transition.

Our solution to this shortcoming is to vary D and to adjust the other four parameters so as to put the problem class at the phase transition—that is, at the point where half the instances in the class are satisfiable. But for any given value of D many combinations of values for N, C, V and L put the problem class at the phase transition. We determine the values of these parameters by keeping D^V, the size of the search space, at a fixed value for all problem instances and then requiring all clauses to have the same "constrainedness", as measured by $(L/D)^V$. Mimicking Boolean 3CNF, we set V to three and aim to keep $(L/D)^V$ at 1/8, which is achieved by setting L to $.5D$. This follows the constant length model advocated by Mitchell, Selman and Levesque [1992]. Finally C is set to whatever value puts the problem class at the phase transition, which we determined experimentally. Thus, while varying D, the parameters that we are holding constant are search space size, the number of variables constrained by each clause and the amount of constraint placed on each, and the proportion of instances with that parameter setting that are solvable.

Our experiments were conducted on domain sizes of 2, 4, 8, 16 and 32. Following the model above we kept D^V at a constant value, 2^{60}. Then for each domain size, we experimentally located the point of the phase transition. The leftmost column of Figure 2 shows the value of C (number of clauses) at which these experiments located the phase transitions.

With the parameters of the random formulas determined, at each domain size we generated a series of random formulas according to the above method and, using a non-Boolean version of the Davis-Putnam procedure, kept the first 25 satisfiable ones.[6] At each domain size 101 attempts were made to solve the suite of 25 formulas with the NB, UU and UB methods. The longer version of this paper proves that a positive CNF nb-formula can be transformed to a b-formula without the ALO formulas. Since our randomly-generated formulas are positive, the two unary encodings that are used here contain only kernels and AMO formulas.

The BB method was not attempted since the binary transformation of these random nb-formulas is unreasonably long. As discussed at the end of Section 4, if the binary transform is used to produce a CNF b-formula from a positive CNF nb-formula, the size of the resulting b-formula is exponential in the clause length of the original nb-formula—which in this case is $\frac{3}{2}D$.

The results of these experiments, shown in Figure 2, consistently follow a clear pattern. In terms of both time and number of flips, all methods show a decline in performance as domain size grows. However, the decline of UU and UB is so sharp that UB is ineffective on domain size 16 and UU is two orders of magnitude slower than NB on domain size 32. On these random formulas, as on the graph colouring problems, the fliprates of all methods decrease as the domain size increases. However, on the random problems, unlike the colouring problems, NB's fliprate suffers the least from increasing domain size. Indeed, by the time domain size reaches 8, NB has a higher fliprate than UU, and this advantage grows as domain size increases. The sharp decline in the UU fliprate results from the rapid increase in the size of the UU formulas, which occurs because the size of their AMO component grows quadratically with domain size.

6 Discussion and Conclusions

The literature reports a number of cases where the transformation approach has been used to attack a non-Boolean problem with Boolean SLS. Such problems include the n-queens problem [Selman *et al.*, 1992], graph colouring [Selman *et al.*, 1992; Hoos, 1998, Chp. 4], the all-interval-series problem [Hoos, 1998, Chp. 4], the Hamiltonian circuit problem

[6] A Boolean Davis-Putnam procedure is incapable of solving Boolean encodings of the instances with larger domain sizes!

Problem	method	P_{noise} setting	formula size	flip rate (flips/sec)	median flips	median time (ms)
Domain size 2	NB	.586	783	199,000	471	2.36
60 vars	UU	.550	903	324,000	3,280	10.1
261 clauses	UB	.440	1023	422,000	8,040	19.1
Domain size 4	NB	.431	1,680	131,000	699	5.36
30 vars	UU	.354	2,040	153,000	12,600	82.9
280 clauses	UB	.156	2,040	262,000	35,800	137
Domain size 8	NB	.332	3,528	80,800	1,050	13.0
20 vars	UU	.044	4,648	75,900	17,800	235
294 clauses	UB	.0524	4,160	193,000	431,000	2230
Domain size 16	NB	.225	7,248	49,500	3,136	63.4
15 vars	UU	.018	10,848	35,600	98,900	2,780
302 clauses	UB		8,448			
Domain size 32	NB	.191	14,736	24,700	5,430	220
12 vars	UU	.010	26,640	15,400	599,000	38,900
307 clauses	UB		17,040			

Figure 2: Results, to three significant figures, for a suite of 25 satisfiable non-Boolean CNF formulas chosen at random. The values recorded in the flips and time column are the flips and time required to solve a single formula, not the entire suite. On those rows missing entries, the median number of flips to solution of 101 runs at noise levels .01, .02 and .03 all exceeded 5,000,000.

[Hoos, 1999; 1998, Chp. 7], random instances of the (finite, discrete) constraint satisfaction problem [Hoos, 1999; 1998, Chp. 7], and STRIPS-style planning [Kautz and Selman, 1992; Kautz *et al.*, 1996; Kautz and Selman, 1996; Ernst *et al.*, 1997; Hoos, 1998, Chp. 4]. Yet, in spite of this widespread use of the transformation approach, only two previous studies have compared the performance of SLS on different encodings of non-Boolean variables: Ernst, Millstein and Weld [1997] compare unary/unary and binary encodings of STRIPS-style planing problems and Hoos [1999; 1998, Chapter 7] compares the unary/unary and binary encodings of random constraint satisfaction problems and of Hamiltonian circuit problems. Other than these two comparative studies, we know of no work that uses Boolean encodings other than unary/unary.

Although Hoos' experiments use a clever methodology designed to provide insight into the structure of SLS search spaces, his study examines only problem instances whose non-Boolean formulation involves variables with domain size 10. And though the work of Ernst, Millstein and Weld is also enlightening in many respects, one cannot determine the domain sizes of the non-Boolean variables in their encodings since they do not report the identity of their suite of planning instances. In contrast to all previous studies, ours considers the effect of varying domain size and hence is able to confirm the expectation that domain size is generally a critical parameter. In particular, our study suggests that Hoos' conclusion—which also appears to be the dominant view in the community— that UU is a viable alternative to NB is generally true only for problems with smaller domain sizes. On problems that require AMO formulas and have large domain sizes our study shows that NB tends to be the best choice and is sometimes the only feasible choice.

In almost all work, when non-Boolean problems are encoded as Boolean formulas the encoding is done in a single stage, directly mapping the problem to a Boolean formula. In some cases this is done manually and in others it is done automatically. In contrast, we advocate a two-stage approach in which the problem is first mapped (manually or automatically) to a non-Boolean formula which is then systematically (and, in our case, automatically) transformed to a Boolean formula. We conjecture that all Boolean encodings that have so far been produced by the single stage approach could be produced by the two stage approach.

It has recently come to our attention that work by Béjar and Manyà [1999] in the area of multi-valued logic has independently led to the development of another generalisation of Walksat, called Regular-WSAT, that appears to be almost identical to NB-Walksat. Regular-WSAT operates on a special class of signed clauses called regular signed clauses. A signed clause is a syntactic variant of an NB-clause except that every variable must be associated with the same domain. To define regular signed clauses, the domain values first must be totally ordered. (In certain other settings, the order is only required to be a lattice.) A signed clause is said to be regular if, in effect, it is positive and for each literal X/d it contains it also contains either every literal X/d' such that $d' < d$ or every literal X/d' such that $d < d'$.

Though Regular-WSAT is presented as a procedure for regular signed clauses, it appears that it would work on any signed clause and thus is akin to NB-Walksat. Nor does there appear to be anything inherent in the presentation of Regular-WSAT that prevents it from handling variables with different domain sizes. However, when operating on a clause containing variables with differing domain sizes it appears that Regular-WSAT and NB-Walksat use different distributions in choosing which literal to flip. Regular-WSAT first chooses a variable from among those in the clause and then chooses one of the values that appear with that variable, whereas NB-Walksat simply chooses from among all the literals in the clause.

Though regular signed clauses are a special class of NB-clauses, they can be used effectively to encode a range of problems including graph colouring problems and round-robin scheduling problems [Béjar and Manyà, 1999]. The extra restrictions placed on regular signed clauses can be exploited to translate them to Boolean clauses more compactly than can NB-clauses [Béjar *et al.*, 2001]. However, we are not aware of any attempt to exploit these translations to solve problems encoded as regular signed clauses.

It would be worthwhile to extend the comparisons made in this paper to include additional solution methods employing SLS. As just suggested, the comparisons could include Boolean translations of regular signed clauses. It would be especially worthwhile to consider SLS methods, such as min-conflicts [Minton *et al.*, 1992], that operate directly on the

original problem or on a constraint satisfaction encoding of the problem.[7]

Of the many questions remaining to be addressed by future work the biggest challenge facing the study of problem encodings—including all encoding issues, not just the handling of non-Boolean variables—is the quest for generality. What can we say about about encoding issues that can guide us in producing effective encodings of new problems? This challenge must be decomposed if progress is to be made. This paper's biggest contribution towards this end is separating out the issue of non-Boolean variables and identifying domain size as a critical parameter.

Finally we claim that, in addition to developing new problem encodings, the applicability of SLS technology can be extended by enriching the language on which the SLS operates. This claim is supported by recent results on pseudo-Boolean constraints [Walser, 1997] and non-CNF formulas [Sebastiani, 1994]. Our success with NB-Walksat adds further weight to the claim.

Acknowledgements

We are grateful to Henry Kautz and Bram Cohen for providing their Walksat code, from which the NB-Walksat implementation was developed. We also thank Peter Stock, who used his non-Boolean Davis-Putnam program to filter out the unsatisfiable instances from our suite of randomly-generated nb-formulas. Thanks to Toby Walsh for many illuminating discussions and to Steve Minton, Bart Selman and the anonymous referees for useful suggestions.

References

[Béjar and Manyà, 1999] Ramon Béjar and Felip Manyà. Solving combinatorial problems with regular local search algorithms. In Harald Ganzinger and David McAllester, editors, *Proc. 6th Int. Conference on Logic for Programming and Automated Reasoning, LPAR, Tbilisi, Georgia*, volume 1705 of *Lecture Notes in Artificial Intelligence*, pages 33–43. Springer-Verlag, 1999.

[Béjar *et al.*, 2001] Ramon Béjar, Reiner Hähnle, and Felip Manyá. A modular reduction of regular logic to classical logic. In *Proc. 31st International Symposium on Multiple-Valued Logics, Warsaw, Poland*. IEEE CS Press, Los Alamitos, May 2001.

[Brafman and Hoos, 1999] Ronen I. Brafman and Holger H. Hoos. To encode or not to encode — I: Linear planning. In *Proc. of the Sixteenth Int. Joint Conf. on Artificial Intelligence*, pages 988–993. Morgan Kaufman, 1999.

[Ernst *et al.*, 1997] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic SAT-compilation of planning problems. In *Proc. of the Fifteenth Int. Joint Conf. on Artificial Intelligence*, pages 1169–1176, 1997.

[Frisch and Peugniez, 1998] Alan M. Frisch and Timothy J. Peugniez. Solving non-Boolean satisfiability problems with local search. In *Fifth Workshop on Automated Reasoning: Bridging the Gap between Theory and Practice*, St. Andrews, Scotland, April 1998.

[Hoos, 1998] Holger H. Hoos. *Stochastic Local Search—Methods, Models, Applications*. PhD thesis, Technical University of Darmstadt, 1998.

[Hoos, 1999] Holger H. Hoos. SAT-encodings, search space structure, and local search performance. In *Proc. of the Sixteenth Int. Joint Conf. on Artificial Intelligence*, pages 296–302. Morgan Kaufman, 1999.

[Kautz and Selman, 1992] Henry A. Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proc. of the Tenth European Conf. on Artificial Intelligence*, pages 359–363, Vienna, Austria, August 1992. John Wiley & Sons.

[Kautz and Selman, 1996] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of the Thirteenth National Conf. on Artificial Intelligence*, pages 1202–1207, Portland, Oregon, USA, 1996. AAAI Press / The MIT Press.

[Kautz *et al.*, 1996] Henry A. Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Fifth Int. Conf.*, pages 374–385, San Francisco, 1996. Morgan Kaufmann.

[Minton *et al.*, 1992] Steven Minton, Mark D. Johnson, 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.

[Mitchell *et al.*, 1992] David Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions of SAT problems. In *Proc. of the Tenth National Conf. on Artificial Intelligence*, pages 459–465, San Jose, CA, 1992. MIT Press.

[Peugniez, 1998] Timothy J. Peugniez. Solving non-Boolean satisfiability problems with local search. BSc thesis, Department of Computer Science, University of York, March 1998.

[Sebastiani, 1994] Roberto Sebastiani. Applying GSAT to non-clausal formulas. *Journal of Artificial Intelligence Research*, 1:309–314, 1994.

[Selman *et al.*, 1992] Bart Selman, Hector J. Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proc. of the Tenth National Conf. on Artificial Intelligence*, pages 440–446. AAAI Press, 1992.

[Selman *et al.*, 1994] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proc. of the Twelfth National Conf. on Artificial Intelligence*, pages 337–343, Menlo Park, CA, USA, 1994. AAAI Press.

[Walser, 1997] Joachim P. Walser. Solving linear pseudo-Boolean constraint problems with local search. In *Proc. of the Fourteenth National Conf. on Artificial Intelligence*, pages 269–274. AAAI Press, 1997.

[7] Brafman and Hoos [1999] have performed a comparison of this type for linear planning problems.

SEARCH, SATISFIABILITY, AND CONSTRAINT SATISFACTION PROBLEMS

CONSTRAINT SATISFACTION PROBLEMS

Backtracking Through Biconnected Components of a Constraint Graph

Jean-François Baget
LIRMM
161, rue Ada
34392 Montpellier, Cedex 5
France
E-mail: baget@lirmm.fr

Yannic S. Tognetti
LIRMM
161, rue Ada
34392 Montpellier, Cedex 5
France
E-mail: tognetti@lirmm.fr

Abstract

The algorithm presented here, BCC, is an enhancement of the well known *Backtrack* used to solve constraint satisfaction problems. Though most backtrack improvements rely on propagation of local informations, BCC uses global knowledge of the constraint graph structure (and in particular its biconnected components) to reduce search space, permanently removing values and compiling partial solutions during exploration. This algorithm performs well by itself, without any filtering, when the biconnected components are small, achieving optimal time complexity in case of a tree. Otherwise, it remains compatible with most existing techniques, adding only a negligible overhead cost.

1 Introduction

Constraint satisfaction problems (CSPs), since their introduction in the early 60's, have flourished in many branches of Artificial Intelligence, and are now used in many "real-life" applications. Since the satisfiability of a CSP is a NP-complete problem, much effort have been devoted to propose faster algorithms and heuristics. Backtrack can be seen as the backbone for those improvements: this algorithm first extends a partial solution by assigning a value to a variable, and undo a previous assignment if no such value can be found.

Backtracking heuristics try to restrict search space, their goal can be to maximize the probability of a "good guess" for the next assignment (variable and value orderings), or to recover more efficiently from a dead-end (backjumping and its variants). *Filtering techniques* use propagation of some local consistency property ahead of the current partial solution, effectively reducing variable domains; they can be used either in a preprocessing phase, or dynamically during exploration. *Structure-driven* algorithms emerged from the identification of tractable classes of CSP, such as trees [Mackworth and Freuder, 1985]; they often transform the CSP into one that can be solved in polynomial time [Gottlob *et al.*, 1999].

The algorithm presented, BCC, does not fit easily in this classification. It accepts any preprocessed ordering of the variables, as long as it respects some properties related to the biconnected components of the constraint graph. Then it exploits the underlying tree structure to reduce *thrashing* during the backtrack, storing partial solutions and removing permanently some values. Though no kind of local consistency property is used, and no look-ahead is performed, BCC performs well when the graph contains small biconnected components, achieving optimal time complexity in case of a tree.

After basic definitions about CSPs, we present in Sect. 3 the version of *Backtrack* used as the kernel of BCC. Sect. 4 is devoted to definitions of the variable orderings compatible with BCC. The algorithm is presented in Sect. 5, along with a proof of its soundness and completeness and an evaluation of its worst-case complexity. In Sect. 6, we compare BCC with popular backjumping or filtering techniques, pointing out the orthogonality of these approaches and showing that a mixed algorithm is possible. Finally, we discuss limitations of our algorithm, and suggest a method to overcome them.

2 Definitions and Notations

A *binary constraint network* $\mathcal{R}$ over a set of symbols $\mathcal{D}$ consists of a set of *variables* $V = \{x_1, \ldots, x_n\}$ (denoted $V(\mathcal{R})$), each x_i associated to a finite *value domain* $D_i \subseteq \mathcal{D}$, and of a set of *constraints*. A constraint R_{ij} between two variables x_i and x_j is a subset of $D_i \times D_j$. A variable x_i is called *instantiated* when it is assigned a value from its domain. A constraint R_{ij} between two instantiated variables is said *satisfied* if $(\text{value}(x_i), \text{value}(x_j)) \in R_{ij}$. This test is called a *consistency check* between x_i and x_j. A *consistent instantiation* of a subset $X \subseteq V(\mathcal{R})$ is an instantiation of all variables in X that satisfies every constraint between variables of X. The CONSTRAINT SATISFACTION PROBLEM (CSP), given a constraint network $\mathcal{R}$, asks whether there exists a consistent instantiation of $V(\mathcal{R})$, called a *solution* of $\mathcal{R}$.

Without loss of generality, we can consider that constraints are always defined over two different variables, and that there is at most one constraint (either R_{ij} or R_{ji}) between two variables. So the *constraint graph* of a network (associating each variable to a node and connecting two nodes whose variables appear in the same constraint) is a non oriented simple graph without loops. In this paper, we will consider some total order $\leq_V$ on $V(\mathcal{R})$, inducing an orientation of this graph: edges can be seen as oriented from the smallest to the greatest of their ends. According to this orientation, we can define the *predecessors* $\Gamma^-(x)$ and the *successors* $\Gamma^+(x)$ of a node x. Note that $|\Gamma^-(x)|$ is the *width* of x [Freuder, 1982].

3 Back to Backtrack

As the `Backtrack` algorithm (BT) is the kernel of BCC, we considered as vital to make it as efficient as possible, according to its usual specifications: 1) variables are examined in a fixed, arbitrary ordering; and 2) no information on the CSP is known or propagated ahead of the current variable.

3.1 Algorithm Kernel

Algorithm 1: `BackTrack`($\mathcal{R}$)

```
Data   : A non empty network R.
Result : true if R admits a solution, false otherwise.
computeOrder (N);
level ← 1;
failure ← false;
while ((level ≠ 0) and (level ≠ |V(R)| + 1)) do
    currentVariable ← V[level];
    if (failure) then
        if (hasMoreValues (currentVariable)) then
            failure ← false;
            level ← nextLevel (currentVariable);
        else level ← previousLevel (currentVariable);
    else
        if (getFirstValue (currentVariable)) then
            level ← nextLevel (currentVariable);
        else
            failure ← true;
            level ← previousLevel (currentVariable);
return (level = |V(R)| + 1);
```

As in [Prosser, 1993], BT is presented in its derecursived version: we will later easily control the variable to study after a successful or failed instantiation, without unstacking recursive calls. For the sake of clarity, it is written to solve a decision problem, though a solution could be "read in the variables". This algorithm should consider any variable ordering, so `computeOrder` only builds the predecessor and successor sets according to the implicit order on the variables, and *sorts the predecessors* sets (for soundness of *partial history*). To respect our specifications, if x is the ith variable according to $\leq_V$, `previousLevel`(x) must return $i-1$ and `nextLevel`(x) $i+1$. According to this restriction, our only freedom to improve BT is in the implementation of the functions `getFirstValue` and `hasMoreValues`.

3.2 Computing Candidates

Suppose BT has built a consistent instantiation of $\{x_1, \ldots, x_{i-1}\}$. Values that, assigned to x_i, would make the instantiation of $\{x_1, \ldots, x_i\}$ consistent, are the *candidates* for x_i. Then BT is sound and complete if `getFirstValue` indicates the first candidate, returning `false` when missing (a *leaf dead-end* at x_i, [Dechter and Frost, 1999]), and successive calls to `hasMoreValues` iterate through them, returning `false` when they have all been enumerated (*internal dead-end*). Testing whether a value is a candidate for x is done in at most width(x) consistency checks.

The iterative method: The most natural method to implement `getFirstValue` (resp. `hasMoreValues`) is to iterate through the domain of x_i, halting as soon as a value pass a consistency check with every y in $\Gamma^-(x_i)$, starting at the first domain value (resp. after the last successful value). Both return `false` when the whole domain has been checked. Though in the worst case, the cost of is $\Theta(|D_i|$ width$(x_i))$, values are checked only when needed: this method is good when the CSP is underconstrained or when width(x_i) is small.

The partition refinement method: Suppose $\Gamma^-(x_i) = \{y_1, \ldots, y_k\}$, and note Δ_0 the domain D_i of x_i. Then compute $\Delta_1, \ldots, \Delta_k$ such that Δ_j contains the values of Δ_{j-1} satisfying the constraint R_{ji}. Then Δ_k is the set of candidates for x_i. Though worst case complexity (when constraints are "full") is the same, it is efficient in practice: if $0 < p \leq 1$ is the *constraint tightness* of a random network [Smith, 1996], then the expected size of Δ_{j+1} is $(1-p) \times |\Delta_j|$; so the expected time to compute Δ_k is $|D_i|/p$. Though some computed candidates may not be used, this method performs well when constraints tightness and variables width are high. The method `hasMoreValues` only iterates through this result.

Storing refinement history: A *partial history* is a trace of the refinements computed in the *current branch* of the backtrack tree: so having computed $\Delta_1 \ldots \Delta_k$ and stored these sets as well as the values of y_j used to obtain them, if we backtrack up to y_p and come back again later to x_i, we only have to refine domains from Δ_{p+1} to Δ_k, replacing the previous values. This mechanism is implemented in Alg. 2.

Algorithm 2: `getFirstValue`(x)

```
Data   : A variable x, belonging to a network R where BT has computed
         a consistent instantiation of all variables preceding x. We also
         suppose that x has at least one ancestor.
Result : Stores refinement history, and returns true unless the computed
         set of candidates is empty.
last ← 1;
w ← width(x);
y ← Γ⁻[0];
while (last ≤ w and Δ[last] ≠ ∅ and usedVal[last] = y.value) do
    y ← Γ⁻[last];
    last++;
while ((last ≤ w) and (Δ[last-1] ≠ ∅)) do
    y ← Γ⁻[last-1];
    usedVal[last] ← y.value;
    Δ[last] ← ∅;
    for (v ∈ Δ[last-1]) do
        if ((y.value, v) ∈ constraint(y, x)) then
            Δ[last] ← Δ[last] ∪ {v};
    last++;
success ← (last = w+1) and (Δ[last-1] ≠ ∅);
if (success) then value ← Δ[last-1][0];
return success;
```

Each variable x is assigned a field `last` (denoted, in an OOP style, x.`last` or simply `last` when there is no ambiguity), a field `value` pointing to the current value of x, a vector containing the *sorted* variables in $\Gamma^-(x)$, and two vectors of size width(x)+1: Δ contains the successive refinements, and `usedVal` contains the values of the ancestors used for successive refinements of the Δ_j. Note that Δ[k] contains all values of Δ[k-1] consistent with the assignment of `usedVal`[k] to the variable `ancestors`[k-1]. Δ[0] is initialized with the domain of x. The partial history mechanism adds the same benefit as *Backmarking* (BM) [Gaschnig, 1979]. Though more memory expensive ($\mathcal{O}(|D_i| \times$ width$(x_i))$ space, for each variable x_i) than the usual version of BM, it is more resistant to the jumps forward and backward that will be explicited further in this paper.

4 Variables Orderings Compatible with BCC

Let us now define the particular variable orderings that will be compatible with BCC. Let $V_1, \ldots, V_k$ be ordered subsets of $V(\mathcal{R})$ such that $V(\mathcal{R}) = \cup_{i=1}^{k} V_i$ (their intersection is not necessarily empty), then predset(V_i)= $\cup_{j=1}^{i-1} V_j$. We call an ordering of $V(\mathcal{R})$ *compatible* with this decomposition if, for every subset V_i, for every variable $x \in V_i$, if $x \notin$ predset(V_i), then x is greater than every variable of predset(V_i). Given such an ordering, the *accessor* of V_i is its smallest element.

4.1 Connected Components

A non empty graph G is *connected* if, for every pair of nodes, there is a walk from one node to the other. A connected component of G is a maximum connected subset of $V(G)$.

Now suppose a network $\mathcal{R}$ whose constraint graph admits $k \geq 2$ connected components, namely $V_1, \ldots, V_k$. We call a *CC-compatible ordering* of the variables of $\mathcal{R}$ an ordering of $V(\mathcal{R})$ compatible with some arbitrary ordering of these connected components. Should we launch BT on this network, and should `computeOrder` store such an ordering, a well-known kind of *thrashing* can occur: if BT fails on component V_i (*i.e.* it registers a dead-end at the accessor of V_i), then it will keep on generating every consistent instantiation of the variables in $V_1, \ldots, V_{i-1}$ to repeat the same failure on V_i.

Since the V_i represent *independent subproblems* (a global knowledge on the graph), the usual method is to launch BT on each connected component, stopping the whole process whenever some V_j admits no solution. It could also be implemented in a simple way with a slight modification of the function `previousLevel`: if x is the accessor of a connected component, `previousLevel`(x) must return 0.

Though very simple, this example illustrates well the method we use later: introduce in the variables ordering some "global" properties of the graph, then use them during backtrack. With a CC-compatible ordering, we benefit from the independence of the sub-problems associated to the connected components; with BCC-compatible orderings, we will benefit from "restricted dependence" between these sub-problems.

4.2 Biconnected Components

A graph G is *k-connected* if the removal of any $k-1$ different nodes does not disconnect it. A *k-connected component* of G is a maximum k-connected subgraph of G. A *biconnected component* (or *bicomponent*), is a 2-connected component. Note we consider the complete graph on two vertices biconnected. The *graph of bicomponents* (obtained by representing each bicomponent as a node, then adding an edge between two components if they share a node) is a forest called the *BCC tree* of G. It is computed in linear time, using two depth-first search (DFS) traversals of G [Tarjan, 1972]. Fig. 1 represents a constraint graph and its associated BCC tree.

If we consider the BCC tree as a rooted forest, a *natural ordering* of its nodes (representing bicomponents) is a total order such that children are greater than their parent. For the BCC tree in Fig. 1, choosing A and B as roots, a DFS could give the order $\{A, F, B, G, E, H, I, D, C\}$, and a BFS (breadth-first search) $\{A, F, B, G, H, C, E, I, D\}$. Both DFS and BFS traversals result in a natural ordering.

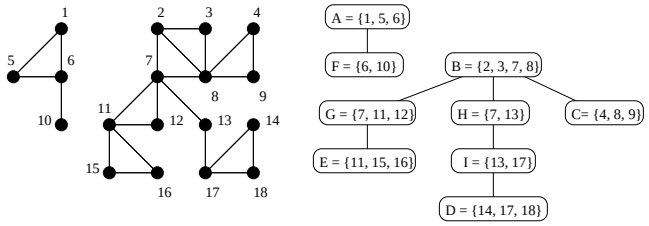

Figure 1: A constraint graph and its BCC tree.

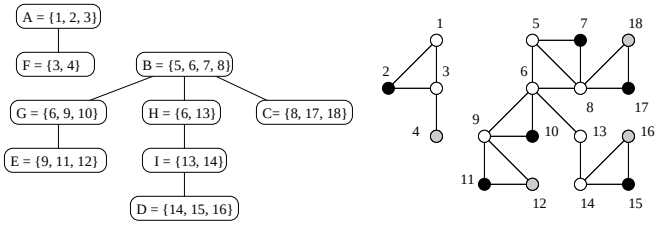

Figure 2: A BCC(DFS, BFS) ordering.

A *BCC-compatible ordering* of the variables of a network $\mathcal{R}$ is an ordering compatible with a natural ordering of its BCC tree (it is also CC-compatible). Such an ordering, if computed by a DFS traversal of the BCC tree, and by a BFS traversal inside the components, would be denoted by a BCC(DFS, BFS) ordering. It can be computed in linear time.

Nodes of the graph in Fig. 2 have been ordered by a BCC(DFS, BFS) ordering. Accessors of a bicomponent have been represented in white. The *second variable* of a bicomponent is the smallest variable in this component, its accessor excepted. By example, in Fig. 2, 9 is the second variable of the component $\{6, 9, 10\}$. Given a BCC-compatible ordering, the *accessor of a variable* is defined as the accessor of the smallest (according to the natural ordering) bicomponent in which it belongs. A second variable is always chosen in the neighborhood of its accessor. Tab. 1 gives the accessors determined by the BCC(DFS, BFS) ordering illustrated in Fig. 2.

Variable	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Accessor	1	1	1	3	5	5	5	5	6	6	9	9	6	13	14	14	8	8

Table 1: Accessors determined by the ordering of Fig. 2.

Suppose a natural ordering of a BCC tree nodes. A *leaf variable* is the greatest variable appearing in a leaf of the BCC tree. In Fig. 2, leaf variables are coloured in grey. We say that a bicomponent C *covers* a leaf variable y if y is the greatest node appearing in the subtree rooted by C (including C itself). Now, we define the *compilers* of a leaf variable y as the accessors of the bicomponents covering y. For the ordering in Fig. 2, compilers are: cmp$(4) = \{1, 3\}$, cmp$(12) = \{6, 9\}$, cmp$(16) = \{6, 13, 14\}$ and cmp$(18) = \{5, 8\}$.

5 The Algorithm BCC

The kernel of BCC is BT, as described in Sect. 3. The call to `computeOrder` orders the variables in a BCC-compatible way, and stores all information on accessors, leaf variables and compilers. This preprocessing takes linear time. We suppose BT is running on a network $\mathcal{R}$.

5.1 Theoretical Foundations

Theorem 1 *If x is the second variable of some bicomponent, any call to* `previousLevel`*(x) can safely (and optimally) return the index of its accessor y.*

Moreover, no solution of $\mathcal{R}$ contains the variable y instantiated with its current value.

A call to `previousLevel`(x) means that we have detected some dead-end (either internal or leaf) at variable x. The first part of this theorem asserts that we can backjump right to the accessor of x, without missing any possible solution, and that it could be dangerous to backjump any further (formal definitions about safe and optimal backjumps can be found, by example, in [Dechter and Frost, 1999]). A dead-end found at variable 13 in Fig. 2 would result in a backjump to variable 6. The second part of the theorem means that we can *permanently remove* the current value of y from its domain.

Proof of Theorem 1: The first part is proved by showing that `previousLevel`(x) performs a particular case of *Graph-Based Backjumping* (GBBJ) [Dechter, 1990]. The accessor y of a second variable x is the only variable in $\Gamma^-(x)$. GBBJ performs a safe backjump to the greatest predecessor of x, we have no other choice than y. It also adds to y a "temporary neighborhood" containing all other predecessors of x (Sect. 6.1). This set being empty, not maintaining it is safe. ◁

Second part relies on the ordering. The following lemma locates the leaf dead-ends responsible for the dead-end at x.

Lemma 1 *Let L be the set of variables that registered a dead-end after the last successful instantiation following x. Then variables in L belong to the subtree T of bicomponents rooted by the smallest component containing x (the* BCC subtree *of x, the accessor of x is also called the accessor of T).*

Sketch of proof: A failure is "propagated upward" in the same bicomponent until a second variable (since variables in the same component, the accessor apart, are numbered consecutively by the compatible ordering). At this point, failure is propagated to the accessor (the backjump in the first part of the theorem), and this accessor belongs to a parent component (a consequence of the natural ordering). ◁

Now suppose there is a solution such that y is assigned the value v, and we prove it is absurd. Let us now consider the subnetwork $\mathcal{R}'$ of $\mathcal{R}$, containing all the variables in the the BCC subtree of x, ordered in the same way as in $\mathcal{R}$. Let us reduce the domain of y in $\mathcal{R}'$ to $\{v\}$. Then $\mathcal{R}'$ admits a solution, and BT will find the first one for the given ordering. Let us now examine the first leaf dead-end (at variable x_e) that caused to bypass this solution in the BT on $\mathcal{R}$ (thanks to Lem. 1, it is a variable of $\mathcal{R}'$). Since we had just before a part of the solution of $\mathcal{R}'$, the domain of x_e has been emptied by a variable outside $\mathcal{R}'$ (or it would not have been a solution of $\mathcal{R}'$). So there is a constraint between x_e and some variable x'_e in a biconnected component outside of $\mathcal{R}'$: this is absurd, since there would be an elementary cycle $x_e, \ldots, x, y, \ldots, r, \ldots, x'_e, x_e$ where r appears in some ancestor of both the component of y and the one of x_e. Then x_e and x'_e would be in the same bicomponent. ◁

Since no variable smaller than y is responsible for the dead-end registered at x, the backjump is optimal. □

Theorem 2 *We suppose* `previousLevel` *has been implemented to match specifications of Th. 1. If x is a leaf variable numbered i, any call to* `nextLevel`*(x), before returning $i+1$, can mark, for every variable y being a compiler of x, that the current value v of y has been compiled and that $i+1$ is the* forward jump *(FJ) for that value (eventually erasing previous FJ for that value).*

Thereafter, every time a call to `nextLevel` *should return the index of a second variable z, if the current value v of its accessor t has been compiled,* `nextLevel` *can safely (and optimally) return instead the forward jump for v.*

Sketch of proof: This is the dual of Th. 1 second part: we had shown that when some failure propagates to the accessor x of a BCC subtree T, then no modification outside T can give a consistent instantiation of T, while keeping x current value. Now, when we have found a consistent instantiation of T, then no modification outside T can make all instantiations of T inconsistent, unless modifying the current value of x. □

A call to `nextLevel` on a leaf means that a whole BCC subtree has been successfully instantiated. In Fig. 2, a call to `nextLevel`(12) means we have a consistent instanciation of the BCC subtree rooted by the bicomponent G covering 12. Before returning 13, this call will store in the accessors of G and E their current values $v(6)$ and $v(9)$, and mark that the FJ for these compiled values is 13. Now we go to 13 and begin to instantiate the subtree containing 6 (already instantiated, and even compiled) and 13, ..., 16. If there is a dead-end at 13, we should backjump to 6 (see Th. 1), and permanently remove its current value $v(6)$ (we must also remove it from its compiled values). Otherwise, it means we can successfully instantiate this subtree until leaf 16, then make a call to `nextLevel`. This call updates all information for compilers of 16, and in particular in 6: the FJ for the compiled value $v(6)$ is now 17. Now suppose we have some dead-end at 17. According to Th. 1, we can backjump to 8. Now suppose further failures make us backtrack to 5, that we successfully instantiate, and that we also instantiate 6 with its compiled value. Should a call to `nextLevel` return 9, it would mean that we found a consistent instantiation of the whole component $\{5, 6, 7, 8\}$. Then we know there is a consistent instantiation of $\{5, \ldots, 16\}$, and we can safely jump forward to 17.

5.2 Implementation

Functions `previousLevel` and `nextLevel` (Alg. 3, 4) naturally encode the above theorems. `nextIndex` and `previousIndex` assume the previous role of `nextLevel` and `previousLevel` (returning $i \pm 1$).

Algorithm 3: `previousLevel`(x)

```
Data   : A variable x.
Result : The index of the variable to be examined by BCC after a dead-end.
if (accessor(x) = x) then return 0;
if (isSecondVariable? (x)) then
    accessor ← accessor(x);
    val ← accessor.value ;
    for (i = 0; i ≤ width(accessor); i++) do
        accessor.Δ[i] ← accessor.Δ[i] \ {val};
    return getIndex (accessor(x));
return previousIndex (x);
```

Algorithm 4: `nextLevel`(x)

```
Data   : A variable x.
Result : The index of the variable to be examined by BCC after a success.

index ← nextIndex(x);
if (isLeaf?(x)) then
    for c ∈ compilers(x) do
        c.compiledValues ← c.compiledValues ∪ {c.value};
        addKey/Value(c.FJ, (c.value, index));

if (index ≠ |V(R)| + 1) then
    nextVariable ← V[index];
    if (isSecondVariable?(nextVariable)) then
        accessor ← accessor(nextVariable);
        if (accessor.value ∈ accessor.compiledValues) then
            index ← getKey/Value(accessor.FJ, accessor.value);

return index;
```

5.3 Worst-Case Complexity

Lemma 2 *Using the BCC algorithm, any BCC subtree is entered at most once for each value in its accessor's domain.*

Proof: A consequence of Th. 1 and 2. Should we enter a BCC subtree T, either we find a consistent instantiation of T, and the current value for its accessor x is compiled (so every time `nextLevel` should return the second variable y for the smallest bicomponent containing x, it jumps instead to the next subtree); or we have registered some dead-end at y (thanks to Lem. 1, this failure came from T), and the current value for the accessor is permanently removed. □

Corollary 1 *When $\mathcal{R}$ is a tree, BCC runs at most in $\mathcal{O}(d^2 n)$, where $n = |V(\mathcal{R})|$ and d is the greatest domain size.*

Proof: Bicomponents are complete graphs with two nodes (accessor and second variable y). They are entered on y, and there will be at most d consistency checks on y. There is n bicomponents, each one entered at most d time (Lem. 2). □

This result is the same as the one given in [Freuder, 1982], obtained by first achieving arc consistency ($\mathcal{O}(d^2 n)$), then by a backtrack-free search. Preprocessing required for BCC is linear, and all possible dead-ends can be encountered, but BCC benefits as much from these failures as from successes. This complexity is optimal [Dechter and Pearl, 1988]. Th. 3 extends this result to any constraint graph.

Theorem 3 *Let n be the number of bicomponents, k be the size of the largest one, and d be the size of the largest domain, then BCC runs in the worst case in $\mathcal{O}(d^k n)$.*

Though this worst-case complexity is the same as [Freuder, 1982], BCC should be more efficient in practice: it backtracks along bicomponents instead of generating all their solutions.

6 BCC and Other Algorithms

Though BCC is efficient when the constraint graph contains numerous bicomponents, it is nothing more than a BT inside these components. We study in this section how to improve BCC with some well-known BT add-ons, and point out that BCC reduces the search tree in an original way.

6.1 Backjumping Schemes

To be sound, BCC must perform a very limited form of Backjumping (BJ) (see Th. 1). The natural question is: what if we use more performant backjumps?

Gaschnig's Backjumping (GBJ) [Gaschnig, 1979] is only triggered in case of a leaf dead-end. It jumps back to the first predecessor of x that emptied its domain. This backjump is safe and optimal in case of a leaf dead-end.

Graph-based Backjumping (GBBJ) [Dechter, 1990] is triggered on any dead-end x, and returns its greatest predecessor y. Safety is ensured by adding to y a temporary set (jumpback set) of predecessors consisting of all the predecessors of x. It is optimal when only graph information is used.

Conflict Directed Backjumping (CDBJ) [Prosser, 1993] integrates GBJ and GBBJ. It backjumps at any dead-end to the greatest variable that removed a value in the domain of x. The jumpback set contains only the predecessors of x that removed some value in x. CDBJ is safe and optimal.

Theorem 4 *Let* `previous`*(x) be a function implementing a safe Backjump scheme XBJ, always returning the index of a variable y belonging to a bicomponent containing x. Then if y is the accessor of x, and if no further backjump from y will return in the component of x, the current value of y can be permanently removed.*

The obtained algorithm BCC+XBJ is sound and complete.

Idea of proof: Restrictions enable to paste proof of Th. 1. □

Corollary 2 *The algorithms BCC+GBJ, BCC+GBBJ and BCC+CDBJ are sound and complete.*

Sketch of proof: See that BCC+GBBJ and BCC+CDBJ respect the specifications in Th. 4, and that a backjump on an accessor y makes us remove a value only if the jumpback set of y only contains its predecessors. Note also that GBJ does not always backjump in the same component (a second variable can backjump to the last element of the parent component, not necessarily the accessor), so BCC specific backjump must be kept to make BCC+GBJ sound and complete. □

Finally, we note that these BJ schemes recover more quickly from a failure than BT, but nothing ensures that this failure will not be repeated over and over. BCC's behaviour cannot be obtained by such a BJ mechanism.

6.2 Filtering Techniques

Different levels of arc consistency can be used in algorithms to prune values that become inconsistent ahead of the last instantiated variable. A constraint R_{ij} is said arc-consistent if, $\forall\ \delta_i \in D_i, \exists\ \delta_j \in D_j\ /\ (\delta_i, \delta_j) \in R_{ij}$, and conversely.

Maintaining Arc-Consistency (MAC) [Sabin and Freuder, 1994] is believed to be the most efficient general CSP algorithm. This family of algorithms rely on maintaining some kind of arc-consistency ahead of the current variable.

Theorem 5 *Let x be the variable currently studied by MAC, and y be a variable whose domain was emptied by the AC phase. Let $C_1, \ldots, C_k$ be the bicomponents belonging to the branch of the BCC tree, between the component C_1 of x and the component C_k of y. Let $z_2, \ldots, z_k$ be their acessors. Then all values from their domains that were not temporarily removed by the AC phase can be* definitively *removed by BCC.*

Sketch of proof: Consider the accessor z_k of the component containing y. If AC propagation has emptied the domain of

y, consider the filtered domain D'_k of z_k. This failure means that, should we extend our current instantiation of all variables until x with any assignment of $\delta \in D'_k$ to z_k, this instantiation does not extend to a solution. Thanks to Th. 1, no instantiation of z_k with the value δ extends to a solution. □

Finally, having proven that BCC and MAC combine their effort to produce a good result, we point out that MAC alone cannot replace BCC's efficiency: perhaps MAC will never consider a value that had to be remove by BCC, but should it consider it, nothing will restrain MAC to fail again on it.

7 Conclusion and Future Works

We have presented here a variation on BT called BCC, whose efficiency relies on the number of bicomponents of the constraint graphs. Early tests have shown that, the number of components increasing, BCC alone quickly compensates its inefficiency inside a component and its results are more than a match for other BT improvements. Moreover, we have shown that BCC itself could be added some well-known BT improvements, such as BJ, FC or MAC, effectively combining the best of two worlds. So an improved BCC could be the perfect candidate for networks whose constraint graph contains many bicomponents.

However, we must confess that such constraint graphs are not so common... So, apart from an original theoretical result expressing that a network whose constraint graph is a tree does not need an AC phase nor a backtrack-free search to be solved in $\mathcal{O}(d^2n)$, what can BCC be used for?

Let $\mathcal{R}$ be a binary constraint network of size n, and x_i and x_j be two of its variables. Then we can transform $\mathcal{R}$ into an equivalent network of $n-1$ variables, by fusioning x_i and x_j into a single variable x_{ij}. The domain of x_{ij} is composed of the pairs of compatible values in the constraint R_{ij} (it is $D_i \times D_j$ when there is no such constraint). Any constraint R_{ki} is then transformed in the following way: if $(\delta_k, \delta_i) \in R_{ki}$, then all pairs $(\delta_k, (\delta_i, \delta))$ such that (δ_i, δ) belongs to the domain of x_{ij} are possible values for the constraint between x_k and x_{ij}. The same construction is performed for constraints incident to x_j. The obtained network is equivalent, but has now $n-1$ variables, and its maximum domain can have now size d^2 (incident constraints can have size d^3). We can iteratively fusion k different nodes, obtaining a network of size $n-k+1$, but where a variable domain can have size d^k.

If a constraint graph admits a k-separator (a set of k nodes whose removal disconnects the graph), then, by fusioning these k variables as indicated above, we create at least two bicomponents sharing the obtained variable. Suppose that we can find in some way separators of size $\leq$ k, the fusion of these separators creating p bicomponents of size $\leq q$. Then BCC (without any improvement), will run in time $\mathcal{O}(d^{kq}p)$. This decomposition may be quite efficient when k is small and the separators cut the graph in a non degenerate way, keeping the size of components comparable. This decomposition method (where polynomial cases are obtained when k and q are both bounded by a constant) is still to be compared to the ones studied in [Gottlob *et al.*, 1999].

To implement and test this decomposition method, we are currently looking for an heuristic that, given an undirected simple graph with weighted edges, find a "small" separator of the graph such that:

(1) removal of the separator creates a graph whose greatest connected component is as small as possible;

(2) product of the weight on edges that belong to the separator is as small as possible.

The weight on edges will be initialized by the constraint tightness. We believe that networks designed by a human being, who decomposes a problem into subproblems slightly related to each other, will be good candidates for this heuristic.

Acknowledgements

We would like to thanks Christian Bessière for his bibliographical suggestions, Michel Chein and Michel Habib who helped making the concept of "semi-independant subgraphs" evolve into the much more formal one of k-connected components, as well as the anonymous referees, for their precious comments and advices.

References

[Dechter and Frost, 1999] R. Dechter and D. Frost. Backtracking Algorithms for Constraint Satisfaction Problems. ICS technical report, University of California, 1999.

[Dechter and Pearl, 1988] R. Dechter and J. Pearl. Network-Based Heuristics for Constraint Satisfaction Problems. *Artficial Intelligence*, 34(1):1–38, 1988.

[Dechter, 1990] R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990.

[Freuder, 1982] E. C. Freuder. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, 29(1):24–32, January 1982.

[Gaschnig, 1979] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Research report CMU–CS–79–124, Carnegie-Mellon University, 1979.

[Gottlob *et al.*, 1999] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. In *Proc. of IJCAI'99*, pages 394–399, 1999.

[Mackworth and Freuder, 1985] A.K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.

[Prosser, 1993] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[Sabin and Freuder, 1994] D. Sabin and E. C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. of ECAI'94*, pages 125–129, 1994.

[Smith, 1996] B. Smith. Locating the Phase Transition in Binary Constraint Satisfaction Problems. *Artificial Intelligence*, 81:155–181, 1996.

[Tarjan, 1972] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. of Computing*, 1:146–160, 1972.

A Constraint Satisfaction Approach to Parametric Differential Equations

Micha Janssen
INGI-UCL
Louvain-La-Neuve, Belgium

Pascal Van Hentenryck
Brown University
Providence, RI

Yves Deville
INGI-UCL
Louvain-La-Neuve, Belgium

Abstract

Initial value problems for parametric ordinary differential equations arise in many areas of science and engineering. Since some of the data is uncertain and given by intervals, traditional numerical methods do not apply. Interval methods provide a way to approach these problems but they often suffer from a loss in precision and high computation costs. This paper presents a constraint satisfaction approach that enhances interval methods with a pruning step based on a global relaxation of the problem. Theoretical and experimental evaluations show that the approach produces significant improvements in accuracy and/or efficiency over the best interval methods.

1 Introduction

Initial value problems for ordinary differential equations arise naturally in many applications in science and engineering, including chemistry, physics, molecular biology, and mechanics to name only a few. An *ordinary differential equation* (ODE) $\mathcal{O}$ is a system of the form

$$\begin{aligned} u_1'(t) &= f_1(t, u_1(t), \ldots, u_n(t)) \\ &\cdots \\ u_n'(t) &= f_n(t, u_1(t), \ldots, u_n(t)) \end{aligned}$$

often denoted in vector notation by $u'(t) = f(t, u(t))$ or $u' = f(t, u)$. In addition, in practice, it is often the case that the parameters and/or the initial values are not known with certainty but are given as intervals. Hence traditional methods do not apply to the resulting parametric ordinary differential equations since they would have to solve infinitely many systems. Interval methods, pioneered by Moore [Moo79], provide an approach to tackle parametric ODEs as they return reliable enclosures of the exact solution at different points in time. Interval methods typically apply a one-step Taylor interval method and make extensive use of automatic differentiation to obtain the Taylor coefficients [Moo79]. Their major problem however is the explosion of the size of the boxes at successive points as they often accumulate errors from point to point and lose accuracy by enclosing the solution by a box (this is called the *wrapping effect*). Lohner's AWA system [Loh87] was an important step in interval methods which features coordinate transformations to tackle the wrapping effect. More recently, Nedialkov and Jackson's IHO method [NJ99] improved on AWA by extending a Hermite-Obreschkoff's approach (which can be viewed as a generalized Taylor method) to intervals. Note that interval methods inherently accommodate uncertain data. Hence, in this paper, we talk about ODEs to denote both traditional and parametric ODEs.

This research takes a constraint satisfaction approach to ODEs. Its basic idea [DJVH98; JDVH99] is to view the solving of ODEs as the iteration of two processes: (1) a *forward* process that computes initial enclosures at given times from enclosures at previous times and bounding boxes and (2) a *pruning* process that reduces the initial enclosures without removing solutions. The real novelty in our approach is the pruning component. Pruning in ODEs however generates significant challenges since ODEs contain unknown functions.

The main contribution of this paper is to show that an effective pruning technique can be derived from a relaxation of the ODE, importing a fundamental principle from constraint satisfaction into the field of differential equations. Three main steps are necessary to derive an effective pruning algorithm. The first step consists in obtaining a relaxation of the ODE by safely approximating its solution using, e.g., generalized Hermite interpolation polynomials. The second step consists in using the mean-value form of this relaxation to solve the relaxation accurately and efficiently. Unfortunately, these two steps, which were skeched in [JDVH99], are not sufficient and the resulting pruning algorithm still suffers from traditional problems of interval methods. The third fundamental step consists in globalizing the pruning by considering several successive relaxations together. This idea of generating a global constraint from a set of more primitive constraints is also at the heart of constraint satisfaction. It makes it possible, in this new context, to address the problem of dependencies (and hence the accumulation of errors) and the wrapping effect simultaneously.[1]

Theoretical and experimental results show the benefits of the approach. The theoretical results show that the pruning step can always be implemented to make our approach

[1] Global constraints in ordinary differential equations have also been found useful in [CB99]. The problem and the techniques in [CB99] are however fundamentally different.

faster than existing methods. In addition, it shows that our approach should be significantly faster when the function f is very complex. Experimental results confirm the theory. They show that the approach often produces many orders of magnitude improvements in accuracy over existing methods while not increasing computation times. Alternatively, at similar accuracy, other approaches are significantly slower. Of particular interest is the fact that our approach is not dependent on high-order Taylor coefficients contrary to other methods.

The rest of the paper is organized as follows. Section 2 introduces the main definitions and notations. Sections 3, 4, and 5 describe the pruning component. Sections 7 and 8 present the theoretical and experimental analyses.

2 Background and Definitions

Basic Notational Conventions Small letters denote real values, vectors and functions of real values. Capital letters denote real matrices, sets, intervals, vectors and functions of intervals. $\mathbb{IR}$ denotes the set of all *closed* intervals $\subseteq \mathbb{R}$ whose bounds are floating-point numbers. A vector of intervals $D \in \mathbb{IR}^n$ is called a *box*. If $r \in \mathbb{R}$, then $\overline{r}$ denotes the smallest interval $I \in \mathbb{IR}$ such that $r \in I$. If $r \in \mathbb{R}^n$, then $\overline{r} = (\overline{r_1}, \ldots, \overline{r_n})$. In this paper, we often use r instead of $\overline{r}$ for simplicity. If $A \subseteq \mathbb{R}^n$, then $\Box A$ denotes the smallest box $D \in \mathbb{IR}^n$ such that $A \subseteq D$. We also assume that $t_0, \ldots, t_k$, t_e and t are reals, $u_0, \ldots, u_k$ are in $\mathbb{R}^n$, and $D_0, \ldots, D_k$ are in $\mathbb{IR}^n$. Finally, we use $\mathbf{t}_k$ to denote $\langle t_0, \ldots, t_k \rangle$, $\mathbf{u}_k$ to denote $\langle u_0, \ldots, u_k \rangle$ and $\mathbf{D}_k$ to denote $\langle D_0, \ldots, D_k \rangle$.

We restrict attention to ODEs that have a unique solution for a given initial value. Techniques to verify this hypothesis numerically are well-known [Moo79; DJVH98]. Moreover, in practice, the objective is to produce (an approximation of) the values of the solution of $\mathcal{O}$ at different points $t_0, t_1, \ldots, t_m$. This motivates the following definition of solutions and its generalization to multistep solutions.

Definition 1 (Solution of an ODE) *The* solution *of an ODE $\mathcal{O}$ is the function $s : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n$ such that $s(t_0, u_0)$ satisfies $\mathcal{O}$ for an initial condition $s(t_0, u_0)(t_0) = u_0$.*

Definition 2 (Multistep solution of an ODE) *The* multistep solution *of an ODE $\mathcal{O}$ is the* partial *function $ms : A \subseteq \mathbb{R}^{k+1} \times (\mathbb{R}^n)^{k+1} \to \mathbb{R} \to \mathbb{R}^n$*

$$ms(\mathbf{t}_k, \mathbf{u}_k)(t) = s(t_0, u_0)(t) \text{ if } u_i = s(t_0, u_0)(t_i)\ (1 \leq i \leq k)$$

where s is the solution of $\mathcal{O}$ and is undefined otherwise.

Since multistep solutions are partial functions, we generalize the definition of interval extensions to partial functions.

Definition 3 (Interval Extension of a Partial Function) *The interval function $G : \mathbb{IR}^n \to \mathbb{IR}^m$ is an* interval extension *of the partial function $g : E \subseteq \mathbb{R}^n \to \mathbb{R}^m$ if*

$$\forall D \in \mathbb{IR}^n : g(E \cap D) \subseteq G(D).$$

where $g(A) = \{g(x) \mid x \in A\}$.

Finally, we generalize the concept of bounding boxes, a fundamental concept in interval methods for ODEs, to multistep methods. Intuitively, a bounding box encloses all solutions of an ODE going through certain boxes at given times.

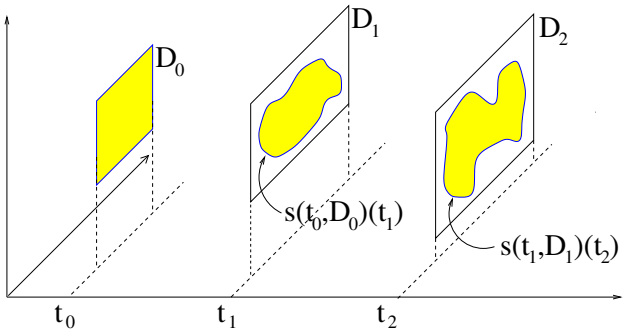

Figure 1: Successive Integration Steps

Definition 4 (Bounding Box) *Let $\mathcal{O}$ be an ODE system, ms be the multistep solution of $\mathcal{O}$, and $\{t_0, \ldots, t_k\} \subseteq T \in \mathbb{IR}$. A box B is a* bounding box *of ms over T wrt ($\mathbf{t}_k$,$\mathbf{D}_k$) if, for all $t \in T$, $ms(\mathbf{t}_k, \mathbf{D}_k)(t) \subseteq B$.*

3 The Constraint Satisfaction Approach

The constraint satisfaction approach followed in this paper was first presented in [DJVH98]. It consists of a generic algorithm for ODEs that iterates two processes: (1) a *forward* process that computes initial enclosures at given times from enclosures at previous times and bounding boxes and (2) a *pruning* process that reduces the initial enclosures without removing solutions. The forward process also provides numerical proofs of existence and uniqueness of the solution. The intuition of the successive integration steps is illustrated in Figure 1. Forward components are standard in interval methods for ODEs. This paper thus focuses on the pruning process, the main novelty of the approach. *Our pruning component is based on relaxations of the ODE as described in the next section.* To our knowledge, no other approach uses relaxations of the ODE to derive pruning operators and the only other approach using a pruning component [NJ99; Rih98] was developed independently.

4 Pruning

The pruning component uses *safe approximations* of the ODE to shrink the boxes produced by the forward process. To understand this idea, it is useful to contrast the constraint satisfaction to nonlinear programming [VHMD97] and to ordinary differential equations. In nonlinear programming, a constraint $c(x_1, \ldots, x_n)$ can be used almost directly for pruning the search space (i.e., the Cartesian products of the intervals I_i associated with the variables x_i). It suffices to take an interval extension $C(X_1, \ldots, X_n)$ of the constraint. Now if $C(I'_1, \ldots, I'_n)$ does not hold, it follows, by definition of interval extensions, that no solution of c lies in $I'_1 \times \ldots \times I'_n$. The interval extension can be seen as a filter that can be used for pruning the search space in many ways. For instance, Numerica uses box(k)-consistency on these interval constraints [VHMD97]. Ordinary differential equations raise new challenges. In an ODE $\forall\, t :\ u' = f(t, u)$, functions u and u' are, of course, unknown. Hence it is not obvious how to obtain a filter to prune boxes.

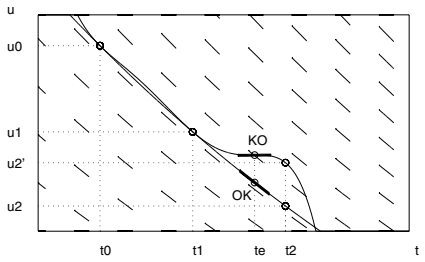

Figure 2: Geometric Intuition of the Multistep Filter

One of the main contributions of our approach is to show how to derive effective pruning operators for parametric ODEs. The first step consists in rewriting the ODE in terms of its multistep solution *ms* to obtain

$$\forall\, t:\ \frac{dms(\mathbf{t}_k,\mathbf{u}_k)}{dt}(t) = f(t, ms(\mathbf{t}_k,\mathbf{u}_k)(t)).$$

Let us denote this formula $\forall\, t:\ fl(\mathbf{t}_k,\mathbf{u}_k)(t)$. This rewriting may not appear useful since *ms* is still an unknown function. However it suggests a way to approximate the ODE. Indeed, we show in Section 6 how to obtain interval extensions of *ms* and $\frac{\partial ms}{\partial t}$ by using polynomial interpolations together with their error terms. This simply requires a bounding box for the considered time interval and safe approximations of *ms* at successive times, both of which are available from the forward process. Once these interval extensions are available, it is possible to obtain an interval formula of the form

$$\forall\, t : FL(\mathbf{t}_k,\mathbf{D}_k)(t)$$

which approximates the original ODE. The above formula is still not ready to be used as a filter because t is universally quantified. The solution here is simpler and consists of restricting attention to a finite set T of times to obtain the relation

$$\forall\, t\ \in\ T : FL(\mathbf{t}_k,\mathbf{D}_k)(t)$$

which produces a computable filter. Indeed, if the relation $FL(\mathbf{t}_k,\mathbf{D}_k)(t)$ does not hold for a time t, it follows that no solution of $u' = f(t,u)$ can go through boxes $D_0,\ldots,D_k$ at times $t_0,\ldots,t_k$. The following definition and proposition capture these concepts more formally.

Definition 5 (Multistep Filters) *Let $\mathcal{O}$ be an ODE and s its solution. A* multistep filter *for $\mathcal{O}$ is an interval relation $FL : \mathbb{R}^{k+1} \times (\mathbb{IR}^n)^{k+1} \rightarrow \mathbb{R} \rightarrow Bool$ satisfying*

$$\left.\begin{array}{c} u_i\ \in\ D_i \\ s(t_0,u_0)(t_i) = u_i\ (0 \le i \le k) \end{array}\right\} \Rightarrow\ \ \forall t:\ FL(\mathbf{t}_k,\mathbf{D}_k)(t).$$

Proposition 1 (Soundness of Multistep Filters) *Let $\mathcal{O}$ be an ODE and let FL be a multistep filter for $\mathcal{O}$. If $FL(\mathbf{t}_k,\mathbf{D}_k)(t)$ does not hold for some t, then there exists no solution of $\mathcal{O}$ going through $\mathbf{D}_k$ at times $\mathbf{t}_k$.*

How can we use this filter to obtain tighter enclosures of the solution? A simple technique consists of pruning the last box produced by the forward process. Assume that D_i is a box enclosing the solution at time t_i $(0 \le i \le k)$ and that we are interested in pruning the last box D_k. A subbox $D \subseteq D_k$ can be pruned away if the condition

$$FL(\mathbf{t}_k, \langle D_0,\ldots,D_{k-1},D\rangle)(t_e)$$

does not hold for some evaluation point t_e. Let us explain briefly the geometric intuition behind this formula by considering what we call *natural filters*. Given interval extensions *MS* and *DMS* of *ms* and $\frac{\partial ms}{\partial t}$, it is possible to approximate the ODE $u' = f(t,u)$ by the formula

$$DMS(\mathbf{t}_k,\mathbf{D}_k)(t) = F(t, MS(\mathbf{t}_k,\mathbf{D}_k)(t)).$$

In this formula, the left-hand side of the equation represents *the approximation of the slope of* u while the right-hand represents *the slope of the approximation of* u. Since the approximations are conservative, these two sides must intersect on boxes containing a solution. Hence an empty intersection means that the boxes used in the formula do not contain the solution to the ODE system. Figure 2 illustrates the intuition. It is generated from an actual ordinary differential equation, considers only points instead of intervals, and ignores error terms for simplicity. It illustrates how this technique can prune away a value as a potential solution at a given time. In the figure, we consider the solution to the equation that evaluates to u_0 and u_1 at t_0 and t_1 respectively. Two possible points u_2 and u_2' are then considered as possible values at t_2. The curve marked KO describes an interpolation polynomial going through u_0, u_1, u_2' at times t_0, t_1, t_2. To determine if u_2' is the value of the solution at time t_2, the idea is to test if the equation is satisfied at time t_e. (We will say more about how to choose t_e later in this paper). As can be seen easily, the slope of the interpolation polynomial is different from the slope specified by f at time t_e and hence u_2' cannot be the value of the solution at t_2. The curve marked OK describes an interpolation polynomial going through u_0, u_1, u_2 at times t_0, t_1, t_2. In this case, the equation is satisfied at time t_e, which means that u_2 cannot be pruned away. The filter proposed earlier generalizes this intuition to boxes. Both the left- and the right-hand sides represent sets of slopes and the filter fails when their intersection is empty. Traditional consistency techniques and algorithms based on this filter can now be applied. For instance, one may be interested in updating the last box produced by the forward process using the operator $D_k = \Box\{r \in D_k \mid FL(\mathbf{t}_k, \langle D_0,\ldots,D_{k-1},r\rangle)(t_e)\}$. The following definition is a novel notion of consistency for ODEs to capture pruning of the last r boxes.

Definition 6 (Forward Consistency of Multistep Filters) *Let FL be a multistep filter. FL is forward(r)-consistent wrt $\mathbf{t}_k$, $\mathbf{D}_k$ and t if*

$$\langle D_{k-r+1},\ldots,D_k\rangle = \Box\{\langle v_1,\ldots,v_r\rangle \in \langle D_{k-r+1},\ldots,D_k\rangle \mid FL(\mathbf{t}_k, \langle D_0,\ldots,D_{k-r},v_1,\ldots,v_r\rangle)(t)\}.$$

The algorithm used in our computational results enforces forward consistency of the filters we now describe.

5 Filters

Filters rely on interval extensions of the multistep solution and its derivative wrt t. These extensions are, in general, based on decomposing the (unknown) multistep solution into

the sum of a computable approximation p and an (unknown) error term e, i.e.,

$$ms(\mathbf{t}_k, \mathbf{u}_k)(t) = p(\mathbf{t}_k, \mathbf{u}_k)(t) + e(\mathbf{t}_k, \mathbf{u}_k)(t).$$

There exist standard techniques to build p, $\frac{\partial p}{\partial t}$ and to bound e, $\frac{\partial e}{\partial t}$. Section 6 reviews how they can be derived from generalized Hermite interpolation polynomials. Here we simply assume that they are available and we show how to use them to build filters.

In the previous section, we mention how natural multistep filters can be obtained by simply replacing the multistep solution and its derivative wrt t by their interval extensions to obtain

$$DMS(\mathbf{t}_k, \mathbf{D}_k)(t) = F(t, MS(\mathbf{t}_k, \mathbf{D}_k)(t)).$$

It is not easy however to enforce forward consistency on a natural filter since the variables may occur in complex nonlinear expressions. In addition, in ODEs, the boxes are often small which makes mean-value forms particularly interesting.

5.1 Mean-Value Filters

The mean-value theorem for a function g states that

$$\exists \xi : x < \xi < m : g(x) = g(m) + g'(\xi)(x - m).$$

A mean-value interval extension G of g in interval I can then be defined as

$$G(X) = G(m) + DG(I)(X - m)$$

for some m in I, where DG is an interval extension of the derivative of g. In order to obtain a mean-value filter, consider the ODE

$$p(\mathbf{t}_k, \mathbf{u}_k)'(t) + e(\mathbf{t}_k, \mathbf{u}_k)'(t) = f(t, p(\mathbf{t}_k, \mathbf{u}_k)(t) + e(\mathbf{t}_k, \mathbf{u}_k)(t))$$

where the multistep solution has been expanded to make the approximations and the error terms explicit. We are thus interested in finding a mean-value interval extension wrt $\mathbf{u}_k$ of the function defined as

$$p(\mathbf{t}_k, \mathbf{u}_k)'(t) + e(\mathbf{t}_k, \mathbf{u}_k)'(t) - f(t, p(\mathbf{t}_k, \mathbf{u}_k)(t) + e(\mathbf{t}_k, \mathbf{u}_k)(t)).$$

However, as mentioned earlier, e is not a computable function and we cannot compute its derivative wrt $\mathbf{u}_k$. Fortunately, it is possible to construct a mean-value filter by considering the error terms as constants and taking a mean-value interval extension of the function $g(\mathbf{t}_k, \mathbf{u}_k, e, de)(t)$ defined as

$$p(\mathbf{t}_k, \mathbf{u}_k)'(t) + de - f(t, p(\mathbf{t}_k, \mathbf{u}_k)(t) + e).$$

If G is a mean-value interval extension of g, then the ODE is approximated by an equation of the form

$$\forall t : G(\mathbf{t}_k, \mathbf{D}_k, E(\mathbf{t}_k, \mathbf{D}_k)(t), DE(\mathbf{t}_k, \mathbf{D}_k)(t))(t) = 0$$

where E and DE are interval extensions of the error term e and of its derivative wrt t. In the following, we call the resulting filter *an implicit mean value filter*.

The implicit mean-value filter for a time t produces an interval constraint of the form

$$A_0X_0 + \ldots + A_kX_k = B$$

where each X_i and B are vectors of n elements and each A_i is an $n \times n$ matrix. Assuming that we are interested in pruning the last box D_k, we obtain an operator as

$$D_k := D_k \cap \Box\{X_k \mid A_0D_0 + \ldots + A_{k-1}D_{k-1} + A_kX_k = B\}.$$

The implicit mean-value filter can be turned into explicit form by inverting the matrix A_k.

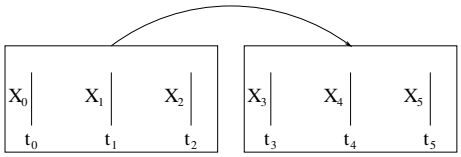

Figure 3: Intuition of the Global Filter

Definition 7 (Explicit Mean-Value Filter) *Let $\mathcal{O}$ be an ODE $u' = f(t, u)$ and rewrite an implicit mean-value filter for $\mathcal{O}$ as*

$$A_0X_0 + \ldots + A_kX_k = B$$

where A_i is an $n \times n$ matrix and B a vector of n elements. An explicit mean-value filter *for $\mathcal{O}$ is given by*

$$X_k = A_k^{-1}B - \textstyle\sum_{i=0}^{k-1} A_k^{-1}A_iX_i.$$

It is simple to make an explicit mean-value filter forward(1)-consistent, since the variables in X_k have been isolated.

5.2 Global Filters

The explicit mean-value filter produces significant pruning of the boxes but it still suffers from a dependency problem typical of interval methods. Consider the application of the filter at two successive time steps of the method. We obtain expressions of the form

$$\begin{array}{lcl} X_k & = & B^1 - M_0^1X_0 + \ldots + M_{k-1}^1X_{k-1} \\ X_{k+1} & = & B^2 - M_1^2X_1 + \ldots + M_k^2X_k \end{array}$$

Note that the second expression considers all tuples of the form $\langle v_1, \ldots, v_k \rangle \in \langle X_1, \ldots, X_k \rangle$ to compute X_{k+1}. But only a subset of these tuples satisfy the first equation for a given $v_0 \in X_0$. The key idea to remove this dependency problem is to use a global filter which clusters a sequence of k mean-value filters together. Figure 3 gives the intuition for $k = 3$. A global filter is then obtained by predicting and pruning k boxes simultaneously. More precisely, the idea is to cluster the k successive filters

$$\begin{array}{lcl} X_k & = & B^1 - M_0^1X_0 + \ldots + M_{k-1}^1X_{k-1} \\ \ldots & & \\ X_{2k-1} & = & B^k - M_{k-1}^kX_{k-1} + \ldots + M_{2(k-1)}^kX_{2(k-1)} \end{array}$$

This filter can be turned into an explicit form by substituting X_k in the second equation, X_k and X_{k-1} in the third equation and so on. The resulting system can be written as a system

$$(X_k \ldots X_{2k-1})^T = B - M(X_1 \ldots X_{k-1})^T$$

where M is a square $nk \times nk$ matrix and B is a vector of nk elements. *It is important to mention that such a global filter is forward(k)-consistent wrt $\mathbf{t}_{2k-1}, \mathbf{D}_{2k-1}$ and t, since the variables $X_k \ldots X_{2k-1}$ have been isolated*. The dependency problem has thus been reduced to the well-known *wrapping effect* which has standard solutions in interval methods.

5.3 The Wrapping Effect

Approximating the set of solutions to an equation by a box may induce a significant loss of precision. This problem is known as the *wrapping effect* and its standard solution is to choose an appropriate coordinate system to represent the solutions compactly. Let us review briefly how to integrate

this idea with our proposals. Consider a global filter that we write as $Z_1 = A_0 Z_0 + B_0$. The idea is to introduce a coordinate transformation $Y_1 = M_1 Z_1$ to obtain a system $M_1 Z_1 = M_1 A_0 Z_0 + M_1 B_0$ where $M_1 A_0$ is diagonally dominant, i.e., the non-diagonal elements are very small compared to the diagonal. If $M_1 A_0$ is diagonally dominant, then the wrapping effect induces no or little loss of precision in computing Y_i. Our pruning step can easily accommodate this idea. Iteration i defines $Y_i = M_i Z_i$ and it sends Z_i, Y_i and M_i to iteration $i+1$ of the main algorithm. Iteration $i+1$ will have to solve the system

$$Z_{i+1} = (A_{i+1} M_i^{-1}) Y_i + B_{i+1}.$$

A coordinate transformation is applied once again to obtain

$$M_{i+1} Z_{i+1} = (M_{i+1} A_{i+1} M_i^{-1}) Y_i + M_{i+1} B_{i+1}$$

so that $M_{i+1} A_{i+1} M_i^{-1}$ is diagonally dominant. The matrices M_i are computed using QR-factorizations [Loh87].

6 Hermite Filters

We now show how to build interval extensions of p, $\partial p/\partial t$, e, $\partial e/\partial t$ by using Hermite interpolation polynomials [SB80]. Informally, a Hermite interpolation polynomial approximates a continuously differentiable function f and is specified by imposing that its values and the values of its successive derivatives at various points be equal to the values of f and of its derivatives at the same points. Note that the number of conditions at the different points may vary.

Definition 8 (Hermite(σ) Interpolation Polynomial) *Let $t_i \in \mathbb{R}, u_i^{(0)} = u_i \in \mathbb{R}^n$ and $u_i^{(j)} = f^{(j-1)}(t_i, u_i)$, $i = 0, \ldots, k$, $j = 0, \ldots, \sigma_i - 1$. Consider the ODE $u' = f(t,u)$. Let $\sigma \in \mathbb{N}^{k+1}, \sigma_i \neq 0$, $i = 0, \ldots, k$, $\sigma_s = \sum_{i=0}^{k} \sigma_i$. The* Hermite($\sigma$) interpolation polynomial *wrt f and $(\mathbf{t}_k, \mathbf{u}_k)$, is the unique polynomial q of degree $\leq \sigma_s - 1$ satisfying*

$$q^{(j)}(t_i) = u_i^{(j)}, \quad j = 0, \ldots, \sigma_i - 1, \;\; i = 0, \ldots, k.$$

It is easy to take interval extensions of a Hermite interpolation polynomial and of its derivatives. The only remaining issue is to bound the error terms. The following standard theorem (e.g., [SB80], [Atk88]) provides the necessary theoretical basis.

Theorem 1 (Hermite Error Term) *Let $p(\mathbf{t}_k, \mathbf{u}_k)$ be the Hermite(σ) interpolation polynomial wrt f and $(\mathbf{t}_k, \mathbf{u}_k)$. Let $u(t) \equiv ms(\mathbf{t}_k, \mathbf{u}_k)(t)$, $T = \square\{t_0, \ldots, t_k, t\}$, $\sigma_s = \sum_{i=0}^{k} \sigma_i$ and $w(t) = \prod_{i=0}^{k} (t - t_i)^{\sigma_i}$. Then, for $i = 1, \ldots, n$,*

- $\exists\, \xi_i \in T : e_i(\mathbf{t}_k, \mathbf{u}_k)(t) = \frac{1}{\sigma_s!} f_i^{(\sigma_s - 1)}(\xi_i, u(\xi_i)) w(t)$;
- $\exists\, \xi_{1,i}, \xi_{2,i} \in T : e_i(\mathbf{t}_k, \mathbf{u}_k)'(t) =$ $\frac{1}{\sigma_s!} f_i^{(\sigma_s - 1)}(\xi_{1,i}, u(\xi_{1,i})) w'(t) + \frac{1}{(\sigma_s+1)!} f_i^{(\sigma_s)}(\xi_{2,i}, u(\xi_{2,i})) w(t)$.

How to use this theorem to bound the error terms? It suffices to take interval extensions of the formula given in the theorem and to replace $\xi_i, \xi_{1,i}, \xi_{2,i}$ by T and $u(\xi_i), u(\xi_{1,i}), u(\xi_{2,i})$ by a bounding box for the ODE over T. In the following, we call *Hermite(σ) filters*, filters based on Hermite(σ) interpolation and we denote a global Hermite(σ) filter by GHF(σ). The following novel result is fundamental and specifies the speed of convergence of Hermite filters. It shows that the order of natural and mean-value Hermite(σ) filters is the sum of the elements in σ, i.e., the number of conditions imposed on the interpolation.

	Cost-1	Cost-2
IHO	–	$2\lfloor \frac{\sigma_s}{2} \rfloor^2 nN_1$ $+O(\sigma_s n N_2)$
GHF	$7k^3n^3$	$((\sigma_m - 1)^2 + 1)knN_1$ $+\sigma_m knN_2$
GHF-1	–	$(\lfloor \frac{\sigma_s - 1}{2} \rfloor^2 + 1)nN_1$ $+O(\sigma_s n N_2)$
GHF-2	$(\frac{7}{8}\sigma_s - \frac{21}{4})\sigma_s^2 n^3$	$(\sigma_s - 2)nN$

Table 1: Complexity Analysis.

Proposition 2 (Order) *Let FL be a natural or mean-value Hermite(σ) filter for ODE $u' = f(t,u)$. Let $t_e - t_k = \Theta(h)$, $t_{i+1} - t_i = \Theta(h)$, $i = 0, \ldots, k-1$, $\sigma_s = \sum_{i=0}^{k} \sigma_i$ and $D_k = \square\{u \in \mathbb{R}^n \mid FL(\mathbf{t}_k, \langle u_0, \ldots, u_{k-1}, u\rangle)(t_e)\}$. Then, under some weak assumptions on f and FL, $w(D_k) = O(h^{\sigma_s + 1})$.*

7 Theoretical Cost Analysis

We now analyse the cost of our method and compare it to Nedialkov's IHO(p,q) method [NJ99], the best interval method we know of. We use the following assumptions. At each step, the forward process uses Moore's Taylor method and the pruning component applies a global Hermite filter together with coordinate transformations (using Lohner's QR-factorization technique). For simplicity of the analysis, we assume that (the natural encoding of) function f contains only arithmetic operations. We denote by N_1 the number of $*, /$ operations in f, by N_2 the number of $\pm$ operations, and by N the sum $N_1 + N_2$. We also assume that the cost of evaluating $\partial f^{(r)}/\partial u$ is n times the cost of evaluating $f^{(r)}$. We define σ_m as $\max(\sigma)$, $\sigma_s = \sigma_0 + \ldots + \sigma_k$, $p+q+1 = \sigma_s$, $q \in \{p, p+1\}$. We also report separately interval arithmetic operations involved in (1) products of a real and an interval matrix (Cost-1) and (2) the generation of Jacobians (Cost-2). Table 7 reports the main cost of a step in the IHO(p,q) method (IHO in the table) and our method GHF(σ) (GHF in the table). It also shows the complexity of two particular cases of GHF(σ). The first case (GHF-1) corresponds to a polynomial with only two interpolation points ($k = 1$, $|\sigma_1 - \sigma_0| \leq 1$), while the second case corresponds to a polynomial imposing two conditions on every interpolation points ($\sigma_0 = \ldots = \sigma_k = 2$). Note that the methods are of the same order in all cases.

The first main result is that *GHF-1 is always cheaper than IHO*, which means that our method can always be made to run faster by choosing only two interpolation points. (The next section will show that substantial improvement in accuracy is also obtained in this case). GHF-2 is more expensive than GHF-1 and IHO when f is simple. However, *when f contains many operations (which is often the case in practical applications), GHF-2 can become substantially faster* because Cost-1 in GHF-2 is independent of f and Cost-2 is substantially smaller in GHF-2 than in GHF-1 and IHO. It also shows the versatility of the approach that can be taylored to the application at hand.

IVP	GHF	IHO	h	Precision			Time	
	σ	p,q		IHO	GHF	Ratio	IHO	GHF
HILB	(2,2,2)	2,3	8E-3	1.8E-1	2.5E-3	72	0.09	0.08
			6E-3	3.8E-4	1.6E-5	24	0.12	0.10
			4E-3	1.6E-5	2.8E-7	57	0.18	0.15
			2E-3	1.5E-7	1.1E-9	136	0.35	0.31
			1E-3	2.0E-9	7.8E-12	256	0.70	0.68
BRUS	(2,2,2)	2,3	1E-1	1.1E-1	3.3E-4	333	0.56	0.55
			7.5E-2	2.2E-4	8.7E-6	25	0.74	0.74
			5E-2	1.1E-5	1.9E-7	58	1.10	1.10
			2.5E-2	1.5E-7	1.1E-9	136	2.20	2.20
BIO1	(2,2,2)	2,3	1.5E-1	8.9E-3	2.5E-4	36	0.16	0.14
			1E-1	2.8E-5	1.0E-6	28	0.23	0.20
			5E-2	2.2E-7	2.8E-9	79	0.44	0.42
			2.5E-2	2.6E-9	1.6E-11	162	0.87	0.87
2BP	(5,5)	4,5	1E-1	3.7E-4	7.2E-6	51	2.10	1.10
			7.5E-2	1.0E-6	1.7E-8	59	2.80	1.70
	(7,6)	6,6	1.25E-1	1.0E-1	3.0E-4	333	2.20	1.40
			1E-1	1.4E-6	9.2E-8	15	2.70	2.00
3BP	(2,2,2,2)	3,4	3E-2	6.9E-2	3.5E-4	197	0.55	0.45
			2E-2	2.1E-4	6.5E-7	323	0.83	0.72
			1E-2	5.3E-8	9.3E-11	570	1.60	1.60
	(5,5,5)	7,7	3E-2	2.4E-2	1.4E-4	171	1.30	0.92
			2E-2	4.4E-7	1.5E-9	293	1.90	1.40
LOR	(3,3)	2,3	1E-2	3.1E+1	4.5E-2	689	2.50	1.90
			7.5E-3	4.0E-1	4.9E-3	82	3.30	2.60
			5E-3	2.5E-2	2.7E-4	93	5.00	3.90
			2.5E-3	3.7E-4	2.0E-6	185	9.90	8.50
BIO2	(4,3)	3,3	6E-3	1.0E-3	4.4E-4	2.3	2.80	2.20
			4E-3	4.0E-6	1.2E-6	3.3	4.10	3.20
			2E-3	1.1E-8	7.3E-10	15	8.30	6.40

Table 2: Experimental Results.

8 Experimental Analysis

We now report experimental results of our C++ implementation on some standard benchmarks [HNW87; Loh87] and two molecular biology problems, which are real-life parametric ODEs given to us by a biologist:

	u_0	$[t_0, t_f]$
Hilbert quadratic problem (HILB):	(2,4)	[0,0.15]
Full Brusselator (BRUS):	(1,2,1)	[0,14]
Two-body problem (2BP):	(1,0,0,1)	[0,24]
Three-body problem(3BP):	(1.2,0,0,-1.05)	[0,1.3]
Lorentz system (LOR):	(15,15,36)	[0,10]
Molecular biology problem (BIO1):	(0.1,0.56,0.14)	[0,4]
Molecular biology problem (BIO2):	(0,0.4,0.3,0.5)	[0,3]

Table 2 compares GHF(σ) and IHO(p,q) methods of the same order. It reports the precision at the last step and execution time (in seconds) of both methods for the same (constant) stepsize. The experimental results follow the same assumptions as in the theoretical analysis. The forward process uses Moore's Taylor method of order $q+1$ (same order as the predictor used in IHO(p,q)) and a Taylor series method of order σ_s to compute a bounding box, except for 2BP and 3BP where we use a series of order 1. The choice of the evaluation time t_e involved in GHF(σ) has not been discussed yet. So far we have no theoretical result about the optimal choice of t_e. We use a simple binary search algorithm to determine a good value for t_e at the beginning of or during the integration. In our experiments, we chose t_e between the last two interpolation points, keeping the distance constant throughout the integration. Our results could be further improved by using a variable distance.

The results indicate that our method produces orders of magnitude improvements in accuracy and runs faster than the best known method. The gain in precision is particularly significant for lower orders. The theoretical results are also confirmed by the experiments. When f contains many operations (e.g. in 3BP), using many interpolation points is particularly effective. For very complex functions, the gain in computation time could become substantial. When f is simple, using few interpolation points becomes more interesting.

As a consequence, we believe that a constraint satisfaction approach to parametric ordinary differential equations is a very promising avenue that complements well traditional approaches. The main open issue at this point is the choice of an optimal evaluation point. This will be the topic of our research in the coming months.

Acknowledgment

This research is partially supported by the *actions de recherche concertée* ARC/95/00-187 and an NSF NYI award. Special thanks to Philippe Delsarte for interesting discussions and for his detailed comments on this paper.

References

[Atk88] K. E. Atkinson *An introduction to Numerical Analysis*. John Wiley & Sons, New York, 1988.

[CB99] J. Cruz, and P. Barahona. An Interval Constraint Approach to Handle Parametric Ordinary Differential Equations for Decision Support. *Proceedings of EKBD-99, 2nd International Workshop on Extraction of Knowledge from Data Bases*, 93-108, 1999.

[DJVH98] Y. Deville, M. Janssen, and P. Van Hentenryck. Consistency Techniques in Ordinary Differential Equations. In *CP'98*, Pisa, Italy, October 1998.

[HNW87] E. Hairer, S.P. Nørsett, G. Wanner. *Solving Ordinary Differential Equations I.* Springer-Verlag, Berlin, 1987.

[JDVH99] M. Janssen, Y. Deville, and P. Van Hentenryck. Multistep Filtering Operators for Ordinary Differential Equations. In *CP'99*, Alexandria, VA, October 1999.

[Loh87] Lohner R. J. Enclosing the solutions of ordinary initial and boundary value problems. In *Computer Arithmetic: Scientific Computation and Programming Languages*, Wiley, 1987.

[Moo79] R.E. Moore. *Methods and Applications of Interval Analysis*. SIAM Publ., 1979.

[NJ99] N.S. Nedialkov and K.R. Jackson. An Interval Hermite-Obreschkoff Method for Computing Rigorous Bounds on the Solution of an Initial Value Problem for an ODE, *Developments in Reliable Computing*, Kluwer, 1999.

[Rih98] R. Rihm. Implicit Methods for Enclosing Solutions of ODEs. *J. of Universal Computer Science*, 4(2), 1998.

[SB80] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, New York, 1980.

[VHMD97] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: a Modeling Language for Global Optimization*. The MIT Press, Cambridge, Mass., 1997.

Improved bounds on the complexity of kB-consistency

Lucas Bordeaux, Eric Monfroy, and Frédéric Benhamou
IRIN, Université de Nantes (France)
{bordeaux, monfroy, benhamou}@irin.univ-nantes.fr

Abstract

kB-consistencies form the class of strong consistencies used in interval constraint programming. We survey, prove, and give theoretical motivations to some technical improvements to a naive kB-consistency algorithm. Our contribution is twofold: on the one hand, we introduce an optimal $3B$-consistency algorithm whose time-complexity of $\mathcal{O}(md^2n)$ improves the known bound by a factor n (m is the number of constraints, n is the number of variables, and d is the maximal size of the intervals of the box). On the other hand, we prove that improved bounds on time complexity can effectively be reached for higher values of k. These results are obtained with very affordable overheads in terms of space complexity.

Keywords: *Strong consistency techniques, Interval constraint propagation.*

1 Introduction

Consistency algorithms are polynomial techniques which aim at struggling against the combinatorial explosion of brute-force search by reducing the search space. Among them, *strong* consistencies [Freuder, 1978] denote generic algorithms which can be tuned to enforce arbitrary intermediate levels of consistency. One of the most (if not *the* most) promising strong consistency techniques is the class of kB-consistencies, where k may vary from 2 to a maximum value corresponding to *global* consistency.

kB-consistency was introduced in [Lhomme, 1993] as a response to the problem called *early quiescence* [Davis, 1987]: when applied to non-trivial, real-valued problems, filtering techniques adapted from arc-consistency do not reduce the search space enough in practice, and stronger consistencies are thus needed to limit the size of the search tree. The basic idea is that given a solver of level $k-1$, each possible value of the variables can be enumerated; if the $k-1$ solver proves the inconsistency of the problem obtained when variable x is instantiated to value v, we know that v is not a possible value for x. This method can be applied to both *domain*-based solvers (by suppressing inconsistent values from domains) and *interval*-based solvers (by tightening bounds).

The *domain-consistency* obtained, called *singleton-consistency* has been emphasized as one of the most efficient techniques for hard finite-domain problems [Debruyne and Bessière, 1997]. This paper deals with the *interval-consistency* obtained, called kB-consistency. kB-consistency has been successfully applied to difficult nonlinear problems, such as a challenging transistor design application [Puget and van Hentenryck, 1998]. The important advantage of kB over path-consistency [Montanari, 1974] and its variants [Dechter and van Beek, 1996; Sam-Haroud and Faltings, 1996] is that it does not require the use of the costly multi-dimensional representations of the constraints needed in these methods.

Let d be the number of elements in the widest initial interval, n be the number of variables, and m be the number of constraints; the known bound of time-complexity for computing kB-consistency is $\mathcal{O}(md^{k-1}n^{2(k-2)})$: $2B$ is computed in $\mathcal{O}(md)$ and each increase in the factor k has a cost of $\mathcal{O}(dn^2)$. Our aim is to show that this complexity is overestimated and that the expected optimum is actually $\mathcal{O}(md^{k-1}n^{k-2})$. We show that this bound can be approached in the general case and that it can be exactly reached in the especially interesting case of $3B$.

This result is obtained by a cautious formulation of the algorithm, in which an optimization pointed out by Lhomme and Lebbah [Lebbah, 1999] is integrated. Informally speaking, each time an interval is reduced, the $k-1$ solver performs an *incomplete* computation. The optimization is to store the result of this computation, so that successive calls to the solver do not need to be restarted from scratch. A dynamic dependency condition is used to determine that some operations can be avoided. We then study the cost of the cumulation of all these incomplete computations to find improved bounds on time complexity.

The outline of this paper is the following: next section introduces the required material and terminology on constraints and solvers. Section 3 introduces interval consistency techniques and in particular kB-consistency. We then define and prove two improvements to a naive algorithm for computing it (Section 4). A theoretical analysis of the resulting algorithm allows us to state the new complexity results in Section 5, while the end of the paper is concerned with conclusions and perspectives.

2 Constraints and Solvers

We note $\mathbb{D}$ the domain of computation. $\mathbb{D}$ is supposed to be *finite* and *totally ordered*: it is thus possible to compute the predecessor and the successor of any element v. This assumption may seem very restrictive, but it corresponds to the usual representation of numbers with computers: in practice, $\mathbb{D}$ is either the set of the canonical floating-point intervals or the subset of the integers encoded on a given number of bits.

2.1 Basic terminology

Let $\mathcal{V}$ be a finite set of *variables*. A *point* t is defined by a coordinate t_x on each variable x, *i.e.,* points are *mappings* from $\mathcal{V}$ to $\mathbb{D}$. A *constraint* c is a set of tuples. For readers more familiar with database-theoretic concepts [Dechter and van Beek, 1996], variables may also be thought of as *field names*, points as *tuples*, and constraints as *relations* or *tables*. Points may also be seen as logical *valuations*.

A CSP $\mathcal{C}$ is a set of constraints. The set of *solutions* to $\mathcal{C}$ is the set $Sol(\mathcal{C})$ of the points which belong to all its constraints, *i.e.,* $Sol(\mathcal{C}) = \bigcap\{c \mid c \in \mathcal{C}\}$. The problem of constraint satisfaction is to find (one/all) solution(s) to a CSP. It is a NP-complete problem in its decisional form.

The *cut* $\sigma_{x=v}(c)$ of a set of points c is the set containing the points of c whose coordinate on x is v: $\sigma_{x=v}(c) = \{t \in c \mid t_x = v\}$. It is also referred to as the *selection* or *instantiation* operator. Among other properties, this operator is *monotonic*: $c \subseteq c' \Rightarrow \sigma_{x=v}(c) \subseteq \sigma_{x=v}(c')$.

Though these definitions are fully general and do not depend on the way constraints are expressed, some points in this paper are concerned with the special case of systems of equations and inequations over $\mathbb{D}$. As an example of such systems, $\{1/x + 1/y = z, x = y\}$ defines the CSP $\mathcal{C} = \{c_1, c_2\}$ where $c_1 = \{\langle x, y, z\rangle \mid 1/x + 1/y = z\}$ and $c_2 = \{\langle x, y, z\rangle \mid x = y\}$. A solution is $\langle x = 2, y = 2, z = 1\rangle$. The cut $\sigma_{z=1}(c_1)$ defines the set $\{\langle x, y, 1\rangle \mid 1/x + 1/y = 1\}$.

Notations: Throughout the text, the considered CSP will be fixed and called $\mathcal{C}$; consequently we shall omit this parameter in most definitions.

2.2 Interval-consistency

We call *search space* (Ω) the set of points among which solutions are searched for. In the interval-consistency framework [Benhamou and Older, 1997], the search space is the cartesian product of *intervals* associated to every variable. The success of this representation comes from the fact that it is a compact and expressive way to approximate the solutions. We note lb_x and ub_x the lower and upper *bounds* of the interval of possible values for variable x. The search space is thus the set containing the points t whose coordinates are inside the bounds ($\forall x \in \mathcal{V},\ t_x \in [lb_x, ub_x]$). It is called a *box*, due to the obvious geometric intuition. When we refer to a bound b_x without further detail, it can be indifferently the lower or the upper bound of x.

An intuitive way to reason on bounds is by relating them to the *facets* of the box. Formally, a facet is the set of points of the box whose coordinate on a given variable is fixed to a bound: it can be written $\sigma_{x=b_x}(\Omega)$ for some variable x and some bound b_x of x.

Notations: Given a box Ω, we note $consistent(\Omega)$ the property $\Omega \cap Sol(\mathcal{C}) \neq \varnothing$, which states that Ω contains a solution.

2.3 Solvers

We define (constraint) *solvers* as functions which verify the following properties:

Definition 1. [Solver] *A solver is a function Φ on boxes such that:*

$$\begin{array}{ll} \Phi(\Omega) \subseteq \Omega & \textit{(contractance)} \\ \Phi(\Omega) \supseteq \Omega \cap Sol(\mathcal{C}) & \textit{(correctness)} \\ \Omega \subseteq \Omega' \ \Rightarrow\ \Phi(\Omega) \subseteq \Phi(\Omega') & \textit{(monotonicity)} \\ \Phi(\Phi(\Omega)) = \Phi(\Omega) & \textit{(idempotence)} \end{array}$$

Note that we deliberately exclude the case of non-idempotent solvers, which are not relevant here. Solvers are intended to reduce the search space; the key-point is that they are usually integrated within a *branch and prune* algorithm, which is guaranteed to find the solutions only if the solver does not remove any of them, hence we call this essential property *correctness*. The opposite property to *correctness* is *completeness*, which states the implication $\neg consistent(\Omega) \Rightarrow \Phi(\Omega) = \varnothing$. Complete solvers are able to detect all inconsistencies, but are usually too costly to be considered practically.

3 kB-consistency

The solvers we now consider can be seen as functions which tighten the intervals by modifying each bound in turn; we often refer to this process as *facet reduction*, meaning that facets are "pushed down". Solvers are defined via *local consistency* properties, which characterize the facets obtained when the reduction stops. We now define kB-consistency inductively, starting with a basic consistency technique for the case $k = 2$.

3.1 2B-consistency

$2B$-consistency is also called hull-consistency by some authors [Benhamou and Older, 1997]. On finite domains, it is the core of some of the main constraint logic programming solvers [Van Hentenryck *et al.*, 1998; Codognet and Diaz, 1996], and it is often referred to as *partial lookahead*.

Definition 2. [2B-consistency] *A box Ω is 2B-consistent if for every bound b_x we have:*

$$\forall c \in \mathcal{C},\ \sigma_{x=b_x}(\Omega) \cap c \neq \varnothing$$

The $2B$-consistency solver is the function Φ_2 which maps a box Ω to the greatest $2B$-consistent box included in Ω.

This definition means that bounds are reduced until each facet of the box intersects every constraint. It can be proved that Φ_2 is a valid function: the computed greatest box exists and is unique; this result comes from more general properties of *confluence* of fixpoint computations [Apt, 1999]. Furthermore, Φ_2 is a valid solver: *correctness* holds because a value v is suppressed from a variable x if $\exists c \in \mathcal{C},\ \sigma_{x=v}(\mathcal{C}) \cap c = \varnothing$, so that we have $\neg consistent(\sigma_{x=v}(\mathcal{C}))$, which proves that the solver only removes facets which do not contain solutions; *idempotence* is straightforward and *monotonicity* comes from the monotonicity of the selection operator.

$2B$-consistency can only be computed for some special constraints called *primitive*, which correspond to the basic arithmetic relations defined by the language. Solvers like Prolog IV decompose arbitrary constraints (*e.g.*, $(2+x).y = z$) into such primitive constraints ($2+x = x', x'y = y', y' = x$).

3.2 kB-consistency

The condition that every facet intersects every constraint is a very rough approximation of (global) consistency. kB-consistency relies on the observation that ideally, a facet should be reduced until it contains a solution; since deciding whether a facet contains a solution is itself a constraint satisfaction problem, a solver Φ_{k-1} can do for this task.

Definition 3. [kB-consistency] *A box Ω is kB-consistent (for $k \geq 3$) if for every bound b_x we have:*

$$\Phi_{k-1}(\sigma_{x=b_x}(\Omega)) \neq \varnothing$$

The kB consistency solver is the function Φ_k such that the box $\Phi_k(\Omega)$ is the greatest kB-consistent box included in Ω.

Informally speaking, bounds are reduced until no facet is rejected by the solver Φ_{k-1}. This definition is recursive: $3B$-consistency uses the $2B$-consistency solver and can in turn be used to define $4B$-consistency, and so on. Properties of *confluence*, *idempotence* and *monotonicity* hold for Φ_k. Its *correctness* is a consequence of the correctness of Φ_{k-1}: since $\Phi_{k-1}(\sigma_{x=b_x}(\Omega)) = \varnothing$ implies $\neg consistent(\mathcal{C})$, the removed facets do not contain any solution. It is easy to prove that Φ_k is "stronger" than Φ_{k-1}, *i.e.*, that $\Phi_k(\Omega) \subseteq \Phi_{k-1}(\Omega)$ (*this is due to the monotonicity of* Φ_{k-1}).

Note that the idea of kB-consistency can be customized and that any consistency technique can be used instead of $2B$-consistency. Some authors have considered the use of Box-consistency [Puget and van Hentenryck, 1998] or *lazy* forms of arc-consistency [Debruyne and Bessière, 1997].

4 Algorithms

We now present the algorithms for computing kB-consistency. As far as $2B$-consistency is concerned, we only sketch the notions required to understand the subsequent proofs, and the reader is referred to [Benhamou and Older, 1997; Lhomme, 1993] for further explanations.

We then give an incremental description of the algorithm for kB-consistency in the general case: starting from a naive algorithm, we successively describe two optimizations used by the experts of this technique and suggested in [Lebbah, 1999] (see also [Delobel, 2000]). The theoretical interest of these improvements shall be studied in the next section.

4.1 Computing 2B-consistency

The algorithm for enforcing $2B$-consistency is a variant of the seminal propagation algorithm - AC3. It is a "constraint-oriented" algorithm, which handles a set of constraints to be considered (generally stored as a queue). It relies on the fact that every constraint c relates a given number of variables - we say that c *depends on* this variable x, or alternatively that c *constrains* x. Figure 1 describes the algorithm.

Two points are essential to the complexity study:

```
1 Q := C
2 while Q ≠ ∅ do
3   Q := Q − {c} % arbitrary choice of c
4   for each variable x constrained by c do
5     reduce lb_x while σ_{x=lb_x}(Ω) ∩ c = ∅
6     reduce ub_x while σ_{x=ub_x}(Ω) ∩ c = ∅
7     if [lb_x, ub_x] has been modified then
8       Q := Q ∪ {c' ∈ C | c' depends on x, c' ≠ c}
```

Figure 1: The algorithm for $2B$-consistency

- In the case of numeric constraints, each constraint depends on (at most) 3 variables;
- Lines 5 and 6 of the algorithm can be performed in bounded time using interval computations.

4.2 Naive algorithm for kB-consistency

The filtering of a box Ω by kB-consistency is computed by reducing each of its facets in turn. In the following algorithm (Figure 2) we use the notation `reduce ... while` to represent the computation of the new bound. This notation actually represents a loop: while the specified condition does not hold, the bound is iteratively reduced to the next (when reducing a lower bound) or previous (upper bound) element. This loop may indeed be optimized, for instance by trying to remove several values at once, or even by using recursive splitting. Unfortunately, these heuristics do not improve the worst-case complexity: at worst, every value has to be considered in turn to be effectively reduced.

```
1 do
2    for each bound b_x do
3       reduce b_x while Φ_{k−1}(σ_{x=b_x}(Ω)) = ∅
4 while Ω is changed
```

Figure 2: Naive algorithm for kB-consistency

The correctness of this algorithm is straightforward since when the **while** loop stops, all the bounds are kB-consistent.

4.3 Storing information

The most obvious drawback of the preceding algorithm is that each time a facet $\sigma_{x=b_x}(\Omega)$ is reconsidered, the solver Φ_{k-1} is restarted from scratch to reduce it. A more clever approach is to store the result of this reduction, in order to reuse it in the next steps (Figure 3). The notation `memorize` is a simple assignment, but it is intended to express the fact that the memorized box has already been computed in the previous line.

```
1 for each bound b_x do mem[b_x] := Ω
2 do
3    for each bound b_x do
4       mem[b_x] := Φ_{k−1}(mem[b_x] ∩ Ω)
5       if mem[b_x] = ∅ then
6          reduce b_x while Φ_{k−1}(σ_{x=b_x}(Ω)) = ∅
7          memorize Φ_{k−1}(σ_{x=b_x}(Ω)) into mem[b_x]
8 while Ω is changed
```

Figure 3: kB algorithm with storage

We use a vector mem[1] which associates to every bound the last non-empty box computed while reducing it. When a bound b_x is reconsidered, it is possible to restart the solver from the box $mem[b_x] \cap \Omega$ (*cf.* Line 4), because we have:

$$\Phi_{k-1}(\sigma_{x=b_x}(\Omega)) \;=\; \Phi_{k-1}(mem[b_x] \cap \Omega)$$

Proof. Note that the intersection $mem[b_x] \cap \Omega$ is a way to rewrite $mem[b_x] \cap \sigma_{x=b_x}(\Omega)$ since $mem[b_x]$ contains only points with value b_x on x. Since $mem[b_x] = \Phi_{k-1}(\sigma_{x=b_x}(\Omega'))$ for some $\Omega' \supseteq \Omega$, a direct consequence of monotonicity is that $\Phi_{k-1}(\sigma_{x=b_x}(\Omega)) \subseteq mem[b_x]$. We also have $\Phi_{k-1}(\sigma_{x=b_x}(\Omega)) \subseteq \sigma_{x=b_x}(\Omega)$ (contractance), and thus $\Phi_{k-1}(\sigma_{x=b_x}(\Omega)) \subseteq mem[b_x] \cap \sigma_{x=b_x}(\Omega))$. Using monotonicity and idempotence we prove that $\Phi_{k-1}(\sigma_{x=b_x}(\Omega)) \subseteq \Phi_{k-1}(mem[b_x] \cap \sigma_{x=b_x}(\Omega)) = \Phi_{k-1}(mem[b_x] \cap \Omega)$. The other inclusion $\supseteq$ is straightforward. □

4.4 Avoiding computation

The second optimization relies on the observation that it is not worth recomputing a bound b_x if $mem[b_x] \subseteq \Omega$, because $mem[b_x] \cap \Omega$ is then equal to $mem[b_x]$ which is already a fixpoint of the solver Φ_{k-1}. In the algorithm of Figure 4, this condition is checked in a naive fashion before each bound reduction:

```
1 for each bound b_x do mem[b_x] := Ω
2 do
3    for each bound b_x do
4       if mem[b_x] ⊈ Ω then
5          mem[b_x] := Φ_{k-1}(mem[b_x] ∩ Ω)
6          if mem[b_x] = ∅ then
7             reduce b_x while Φ_{k-1}(σ_{x=b_x}(Ω)) = ∅
8             memorize Φ_{k-1}(σ_{x=b_x}(Ω)) into mem[b_x]
9 while Ω is changed
```

Figure 4: Algorithm with dynamic dependency checkings

Figure 5 is an illustration of the preceding considerations: in this figure, bound ub_x is reduced but the global box still contains $mem[lb_y]$ (= $\Phi_{k-1}(\sigma_{y=lb_y}(\Omega'))$ for some box $\Omega' \supseteq \Omega$). The re-computation of bound lb_x can therefore be avoided.

5 Complexity analysis

This section is concerned with the theoretical study of the algorithms, starting with the special case of $2B$-consistency. We then study the complexity of the algorithms of Section 4, and point out little improvements needed to compute kB in near-optimal time. As a comparison, we first recall the original complexity results given in [Lhomme, 1993]. As usual in the literature [Mackworth and Freuder, 1985], the complexity of local consistency algorithms is expressed as a function of the following parameters:

[1]Technically, bounds may be seen as variables, it is thus legitimate to use their value (c.f. $\sigma_{x=b_x}(\Omega)$) or to modify them (as in "reduce b_x while"). Our use of the notation $mem[b_x]$ is slightly abusive, the intended meaning is that the stored value is associated to the bound, not to its current value.

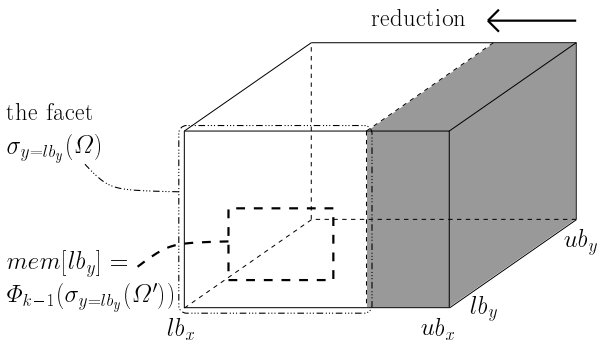

Figure 5: An illustration of the algorithms

- $n = |\mathcal{V}|$ the number of variables,
- $m = |\mathcal{C}|$ the number of constraints,
- d the number of elements in the greatest interval of the box.

In general, it is possible to artificially define as many constraints as is wished on a given number of variables. On the contrary, it is not possible to have $n > 3m$ (for ternary constraints) unless the absurd case of disconnected constraint graphs is accepted, thus $n = \mathcal{O}(m)$.

5.1 Complexity of 2B-consistency

The complexity of $2B$-consistency has been determined by [Lhomme, 1993] using an argument inspired from [Mackworth and Freuder, 1985]. We recall this result and give a simpler proof:

Complexity 1. *The time-complexity for the algorithm computing $2B$-consistency (see Figure 1) is $\mathcal{O}(md)$.*

Proof. Between two insertions of a constraint c into the set $\mathcal{Q}$, the reduction of one of the intervals on which c depends must have occurred. c depends on a bounded number of variables (3 in our case), and each interval may be reduced at most d times. The overall number of insertions of a given constraint into the queue is hence bounded by $\mathcal{O}(d)$, and the number of insertions of the m constraints is $\mathcal{O}(md)$. Since Lines 5 and 6 of the algorithm require constant time, this gives the complexity. □

We also recall the best-case running time: when no reduction is obtained (the box is already $2B$-Consistent), every constraint is dequeued in time $\Theta(m)$.

5.2 Complexity of the original algorithm

The complexity of the original algorithm for kB-consistency presented in [Lhomme, 1993] is the same as the one of the naive algorithm presented in Figure 2. To show that this algorithm runs in time $\mathcal{O}(md^{k-1}n^{2(k-2)})$, we just prove the following result:

Complexity 2. *The number of calls to the solver Φ_{k-1} in the naive kB-consistency algorithm is $\mathcal{O}(dn^2)$.*

Proof. The worst case is obtained when re-computing all facets (Line 2 of the algorithm of Figure 2) results in a minimal reduction (removal of one element) of only one of the bounds: consequently, $\mathcal{O}(n)$ applications of Φ_{k-1} are needed to remove one element. Since one interval is associated to each of the n variables, dn elements may be suppressed one after the other in the worst case, hence the overall number of applications of the solver Φ_{k-1} is $\mathcal{O}(dn^2)$. □

Note that this complexity does not seem to be optimal: since we use the solver Φ_{k-1} to test each element of every variable, we could expect an optimal algorithm to perform only $\mathcal{O}(dn)$ calls to Φ_{k-1}. We shall see how this presumably optimal bound can be reached in the case of $3B$-consistency.

5.3 3B-consistency

We now have all the theoretical elements at hand for a critical study of the algorithm of Figure 4. We first restrict ourselves to the case of $3B$-consistency, whose expected optimal complexity is $\mathcal{O}(md^2n)$. The algorithm of Figure 4 is not optimal yet for a simple reason: to suppress some value on a given interval, it may be needed to apply several $2B$-consistencies, each time with cost $\mathcal{O}(m)$. The problem is that $2B$ is run on all constraints in a systematic way, while only the ones which depend on a bound which has been reduced since the last run should be enqueued.

This situation may be avoided with a finer storage of the queues used by $2B$-consistency (Figure 6): a queue $q[b_x]$ is attached to each facet b_x, and each time the reduction of a facet stops, the corresponding queue is empty. Only the constraints which effectively have to be reconsidered are enqueued (Line 12). In the following the expression `using` means that Φ_2 is run with $q[b_x]$ as an initial queue instead of the queue containing all the constraints used in Figure 1:

```
1  for each bound b_x do mem[b_x] := Ω
2                        q[b_x] := C
3  do
4   for each bound b_x do
5     mem[b_x] := Φ2(mem[b_x]) using q[b_x]
6     if mem[b_x] = ∅ then
7       reduce b_x while Φ2(σ_{x=b_x}(Ω)) = ∅
8       memorize Φ2(σ_{x=b_x}(Ω)) into mem[b_x]
9       for each bound b_y do
10        if mem[b_y] ⊈ Ω then
11          mem[b_y] := mem[b_y] ∩ Ω
12          q[b_y] := q[b_y] ∪ {c | c depends on b_x}
13 while Ω is changed
```

Figure 6: Optimized handling of the queues in $3B$

Note that this algorithm is essentially the same as the one of Figure 4 where the test for inclusion in Line 10 and the intersection in Line 11 are applied in an optimized way. These two operations can be especially easily computed since only the interval for the variable x which has effectively been changed has to be checked or reduced.

Complexity 3. *The algorithm of Figure 6 is optimal ($\mathcal{O}(md^2n)$).*

Proof. Several calls to the solver Φ_2 may be required to suppress a value v from a given variable. Throughout these calls, each constraint is taken into account only if one of the variables on which it depends has been reduced (either during the current computation of $2B$-consistency or between the previous computation and the current one). The cumulated number of constraint insertions into the queue for all the calls related to value v is then bounded by $\mathcal{O}(md)$. Since $\mathcal{O}(dn)$ such values may be suppressed, the global cost is $\mathcal{O}(md^2n)$. □

This bound is correct because the number of tests of interval inclusions has been optimized: in the algorithm, $\mathcal{O}(n)$ interval inclusions are checked every time one of the $\mathcal{O}(dn)$ elements is effectively removed, and the cost of these tests is then $\mathcal{O}(dn^2)$. Since n is $\mathcal{O}(m)$, this complexity is less than $\mathcal{O}(md^2n)$.

5.4 Complexity of kB-consistency

We refine the analysis of the complexity of kB-consistency. Figure 7 shows the savings obtained by the memorization presented before. Ω_1 is a facet of some box. It is reduced by $(k-1)B$-consistency in order to show that the corresponding value v is inconsistent. The values suppressed by $(k-2)B$-consistency appear as tiles on the figure. The second part of the figure shows a case where the search space Ω' has been reduced and the re-computation of $(k-1)B$-consistency is needed.

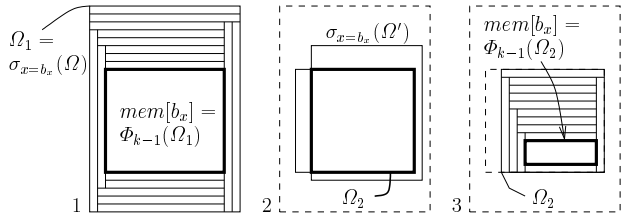

Figure 7: Successive reductions of a facet $\sigma_{x=v}(\Omega)$.

To determine the complexity of kB-consistency, we bound the number of filterings by $(k-2)B$-consistency (i.e. *the tiles*). The keypoint is that the algorithm stores the suppressions obtained by $(k-2)B$, and avoids computing them several times. We have the following result:

Complexity 4. *When computing kB-consistency, the number of computations of $(k-2)B$-consistency is $\mathcal{O}(d^2n^3)$.*

Proof. Consider the computation needed to suppress some value v associated to a variable x (v may be thought of as the bound b_x of the box). $\mathcal{O}(dn)$ such values exist. Due to the memorization, it is needed to re-compute the $(k-1)B$-consistency on the facet $\sigma_{x=v}(\Omega)$ only in the case where one of the values of the corresponding memorized box ($mem[b_x]$) has been removed. In the worst case, it may happen that no computation of $(k-1)B$-consistency reduces the memory; up to $\mathcal{O}(dn)$ such computations can thus occur, since $\mathcal{O}(dn)$ suppressions of a single value of the memory may occur. Each time, the number of $(k-2)B$-consistencies that are checked is $\mathcal{O}(n)$ (one is run on each facet of $\sigma_{x=v}(\Omega)$).

The overall number of $(k-2)B$-consistencies is bounded by $\mathcal{O}(d^2n^3)$. □

Since the original bound on the number of $(k-2)B$-consistencies is $\mathcal{O}(d^2n^4)$, we have shown an improvement of the worst-case behaviour. We summarize the results for $2B$, $3B$, and stronger consistencies ($k' \geq 1$):

$(2k')$B	$\mathcal{O}(md.(d^2n^3)^{k'-1})$
$(2k'+1)$B	$\mathcal{O}(md^2n.(d^2n^3)^{k'-1})$

6 Conclusion

Investigating several improvements to a basic algorithm for computing kB-consistency, we have designed an optimal algorithm for $3B$-consistency, and shown that improved bounds are reached for higher values of k. Note that the algorithms presented here are very affordable in terms of space complexity: the storage of the boxes needed by the algorithms of Figures 4 and 6 requires space $\mathcal{O}(kn^2)$ (n^2 space is needed to store a box for each facet at each level and no two applications of some solver for the same level of k may coexist). The handling of the queues in the algorithm of Figure 6 requires $\mathcal{O}(nm)$ additional space.

In terms of algorithms, our main contribution was to overview and justify theoretically some improvements based on some *memorization of computation*. Though our purpose was not to discuss the practical relevance of this technique, it actually proves to be effective on our implementation, since it allows saving some percents of the computations of facets. Typical benchmarks show that 5% to 60% of the facet reduction may be avoided while computing $3B$-consistency. It might be possible to generalize these memorization techniques to each level of kB-consistency, and hence to improve the theoretical time-complexity - to the detriment of space-complexity.

Our study of the *complexity* of interval consistency techniques may be regarded as complementary to the qualitative results which compare the *filtering* obtained by the algorithms. For instance, it is well-known that $3B$-consistency allows to compute more precise intervals than Box-consistency [Collavizza *et al.*, 1999], but that it is also a more costly technique. Since the efficiency of constraint solving depends on the compromise between the pruning obtained and the time required to achieve it, an interesting perspective is to study the difference between the two algorithms in terms of theoretical costs.

Note that our study was restricted to the original definition of kB-consistency, and that the alternative definition based on Box-consistency introduced in [Puget and van Hentenryck, 1998] has not been considered. Since Box-consistency is computed in a very similar way to $2B$, it seems however that little customization of our algorithm is needed to handle this case. A more challenging task would be to determine whether similar optimizations to the ones presented in this paper may be helpful in the combinatorial case, where *domains* are considered instead of *intervals*.

Acknowledgements *We express our gratitude to Olivier Lhomme for its encouragements and advice.*

References

[Apt, 1999] K. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2), 1999.

[Benhamou and Older, 1997] F. Benhamou and W. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1), 1997.

[Codognet and Diaz, 1996] P. Codognet and D. Diaz. Compiling constraints in `clp(fd)`. *Journal of Logic Programming*, 27(3), 1996.

[Collavizza *et al.*, 1999] H. Collavizza, F. Delobel, and M. Rueher. Comparing partial consistencies. *Reliable Computing*, 5(3), 1999.

[Davis, 1987] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32, 1987.

[Debruyne and Bessière, 1997] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the 15th IJCAI*, Nagoya, Japan, 1997.

[Dechter and van Beek, 1996] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1), 1996.

[Delobel, 2000] F. Delobel. *Résolution de contraintes réelles non-linéaires*. PhD thesis, Université de Nice - Sophia Antipolis, 2000.

[Freuder, 1978] E.C. Freuder. Synthesising constraint expressions. *Communications of the ACM*, 21, 1978.

[Lebbah, 1999] Y. Lebbah. *Contribution à la résolution de contraintes par consistance forte (in french)*. PhD thesis, ENSTI des Mines de Nantes, 1999.

[Lhomme, 1993] O. Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of the 13th IJCAI*, pages 232–238, Chambéry, France, 1993.

[Mackworth and Freuder, 1985] A. Mackworth and E. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1), 1985.

[Montanari, 1974] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2), 1974.

[Puget and van Hentenryck, 1998] J.-F. Puget and P. van Hentenryck. A constraint satisfaction approach to a circuit design problem. *Journal of Global Optimization*, 13, 1998.

[Sam-Haroud and Faltings, 1996] J. Sam-Haroud and B. V. Faltings. Consistency techniques for continuous constraints. *Constraints*, 1, 1996.

[Van Hentenryck *et al.*, 1998] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language `cc(fd)`. *Journal of Logic Programming*, 37(1-3), 1998.

Refining the Basic Constraint Propagation Algorithm[1]

Christian Bessière[2]
LIRMM–CNRS (UMR 5506)
161 rue Ada
34392 Montpellier Cedex 5, France
bessiere@lirmm.fr

Jean-Charles Régin
ILOG
1681, route des Dolines
06560 Valbonne, France
regin@ilog.fr

Abstract

Propagating constraints is the main feature of any constraint solver. This is thus of prime importance to manage constraint propagation as efficiently as possible, justifying the use of the best algorithms. But the ease of integration is also one of the concerns when implementing an algorithm in a constraint solver. This paper focuses on AC-3, which is the simplest arc consistency algorithm known so far. We propose two refinements that preserve as much as possible the ease of integration into a solver (no heavy data structure to be maintained during search), while giving some noticeable improvements in efficiency. One of the proposed refinements is analytically compared to AC-6, showing interesting properties, such as optimality of its worst-case time complexity.

1 Introduction

Constraint propagation is the basic operation in constraint programming. It is now well-recognized that its extensive use is necessary when we want to efficiently solve hard constraint satisfaction problems. All the constraint solvers use it as a basic step. Thus, each improvement that can be incorporated in a constraint propagation algorithm has an immediate effect on the behavior of the constraint solving engine. In practical applications, many constraints are of well-known types for which specific algorithms are available. These algorithms generally receive a set of removed values for one of the variables involved in the constraint, and propagate these deletions according to the constraint. They are usually as cheap as one can expect in cpu time. This state of things implies that most of the existing solving engines are based on a constraint-oriented or variable-oriented propagation scheme (ILOG Solver, CHOCO, etc.). And AC-3, with its natural both constraint-oriented [Mackworth, 1977] and variable-oriented [McGregor, 1979] propagation of the constraints, is the generic constraint propagation algorithm which fits the best this propagation scheme. Its successors, AC-4, AC-6, and AC-7, indeed, were written with a value-oriented propagation. This is one of the reasons why AC-3 is the algorithm which is usually used to propagate those constraints for which nothing special is known about the semantics (and then for which no specific algorithm is available). This algorithm has a second strong advantage when compared to AC-4, AC-6 or AC-7, namely, its independence with regard to specific data structure which should be maintained if used during a search procedure. (And following that, a greater easiness to implement it.) On the contrary, its successors, while more efficient when applied to networks where much propagation occurs ([Bessière *et al.*, 1995; Bessière *et al.*, 1999]), need to maintain some additional data structures.

In this paper, our purpose is to present two new algorithms, `AC2000` and `AC2001`, which, like AC-3, accept variable-oriented and constraint-oriented propagation, and which improve AC-3 in efficiency (both in terms of constraint checks and cpu time). `AC2000`, like AC-3, is free of any data structure to be maintained during search. `AC2001`, at the price of a slight extra data structure (just an integer for each value-constraint pair) reaches an optimal worst-case time complexity.[3] It leads to substantial gains, which are shown both on randomly generated and real-world instances of problems. A comparison with AC-6 shows interesting theoretical properties. Regarding the human cost of their implementation, `AC2000` needs a few lines more than the classical AC-3, and `AC2001` needs the management of its additional data structure.

2 Preliminaries

Constraint network. A finite binary *constraint network* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is defined as a set of n *variables* $\mathcal{X} = \{X_1, \ldots, X_n\}$, a set of *domains* $\mathcal{D} = \{D(X_1), \ldots, D(X_n)\}$, where $D(X_i)$ is the finite set of possible *values* for variable X_i, and a set $\mathcal{C}$ of e binary *constraints* between pairs of variables. A constraint C_{ij} on the ordered set of variables (X_i, X_j) is a subset of the Cartesian product $D(X_i) \times D(X_j)$ that specifies the *allowed* combinations of values for the variables X_i and X_j. (For each constraint C_{ij}, a constraint C_{ji} is defined between (X_j, X_i), allowing the same pairs of values

[1]This work has been partially financed by ILOG under a research collaboration contract ILOG/CNRS/University of Montpellier II.

[2]Member of the COCONUT group.

[3]A related paper by Zhang and Yap appears in these proceedings.

in the reverse order.) Verifying whether a given pair (v_i, v_j) is allowed by C_{ij} or not is called a *constraint check*. A *solution* of a constraint network is an instantiation of the variables such that all the constraints are satisfied.

Arc consistency. Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network, and C_{ij} a constraint in $\mathcal{C}$. A value $v_i \in D(X_i)$ is *consistent with* C_{ij} iff $\exists v_j \in D(X_j)$ such that $(v_i, v_j) \in C_{ij}$. (v_j is then called a *support* for (X_i, v_i) on C_{ij}.) A value $v_i \in D(X_i)$ is *viable* iff it has support in all $D(X_j)$ such that $C_{ij} \in \mathcal{C}$. $\mathcal{P}$ is *arc consistent* iff all the values in all the domains are viable. We achieve arc consistency in $\mathcal{P}$ by removing every value which is not viable.

3 A first stab at improving AC-3

3.1 Background on AC-3

Before presenting the algorithm we propose in this section, let us briefly recall the AC-3 algorithm. We present it with the structure proposed by McGregor [McGregor, 1979], which is built on a variable-based propagation scheme. This will be recommended for the algorithm presented in the next subsection. The main algorithm (see Algorithm 1) is very close to the original AC-3, with an initialization phase (lines 1 to 3), a propagation phase (line 4), and with the use of the function `Revise3`(X_i, X_j) that removes from $D(X_i)$ the values without support in $D(X_j)$ (line 2). But instead of handling a queue of the constraints to be propagated, it uses a queue[4] of the variables that have seen their domain modified. When a variable X_j is picked from the queue (line 2 of `Propagation3` in Algorithm 2), all the constraints involving X_j are propagated with `Revise3`. This is the only change w.r.t. the Mackworth's version of AC-3.[5] (This algorithm has been presented in [Chmeiss and Jégou, 1998] under the name AC-8.)

Algorithm 1: `Main algorithm`

```
function AC (in X: set): Boolean
1  Q ← ∅;
   for each X_i ∈ X do
       for each X_j such that C_ij ∈ C do
2          if Revise-X(X_i, X_j, false[6]) then
               if D(X_i) = ∅ then return false ;
3              Q ← Q ∪ {X_i};
4  return Propagation-X(Q);
```

3.2 The algorithm AC2000

If we closely examine the behavior of AC-3, we see that removing a single value v_j from a domain $D(X_j)$ (inside function `Revise3`) is enough to put X_j in the propagation queue (line 3 of `AC` in Algorithm 1 and line 5 of `Propagation3` in Algorithm 2), and to provoke a call to `Revise3`(X_i, X_j) for every constraint C_{ij} involving X_j (lines 3 and 4 of `Propagation3`). `Revise3` will look for a support for every value in $D(X_i)$ whereas for some of them v_j was perhaps not even a support. (As a simple example, we can take the constraint $X_i = X_j$, where $D(X_i) = D(X_j) = [1..11]$. Removing value 11 from $D(X_j)$ leads to a call to `Revise3`(X_i, X_j), which will look for a support for every value in $D(X_i)$, for a total cost of $1+2+\ldots+9+10+10 = 65$ constraint checks, whereas only $(X_i, 11)$ had lost support.) We exploit this remark in `AC2000`. Instead of looking blindly for a support for a value $v_i \in D(X_i)$ each time $D(X_j)$ is modified, we do that only under some conditions. In addition to the queue Q of variables modified, we use a second data structure, $\Delta(X_j)$,[7] which for each variable X_j contains the values removed since the last propagation of X_j. When a call to `Revise2000`$(X_i, X_j, lazymode)$ is performed, instead of systematically looking whether a value v_i still has support in $D(X_j)$, we first check that v_i really lost a support, namely one of its supports is in $\Delta(X_j)$ (line 3 of `Revise2000` in Algorithm 3). The larger $\Delta(X_j)$ is, the more expensive that process is, and the greater the probability to actually find a support to v_i in this set is. So, we perform this "lazymode" only if $\Delta(X_j)$ is sufficiently smaller than $D(X_j)$. We use a parameter Ratio to decide that. (See line 1 of `Propagation2000` in Algorithm 3.) The Boolean lazymode is set to true when the ratio is not reached. Otherwise, lazymode is false, and `Revise2000` performs exactly as `Revise3`, going directly to lines 4 to 5 of `Revise2000` without testing the second part of line 3.

If we run `AC2000` on our previous example, we have $\Delta(X_j) = \{11\}$. If $|\Delta(X_j)| < \text{Ratio} \cdot |D(X_j)|$, then for each $v_i \in D(X_i)$, we check whether $\Delta(X_j)$ contains a support of v_i before looking for a support for v_i. This requires 11 constraint checks. The only value for which support is effectively sought is $(X_i, 11)$. That requires 10 additional

Algorithm 2: `Subprocedures for AC-3`

```
function Propagation3 (in Q: set): Boolean
1  while Q ≠ ∅ do
2      pick X_j from Q;
3      for each X_i such that C_ij ∈ C do
4          if Revise-X(X_i, X_j) then
               if D(X_i) = ∅ then return false ;
5              Q ← Q ∪ {X_i};
   return true ;

function Revise3 (in X_i, X_j: variable): Boolean
   CHANGE ← false;
   for each v_i ∈ D(X_i) do
       if ∄v_j ∈ D(X_j)/C_ij(v_i, v_j) then
           remove v_i from D(X_i);
           CHANGE ← true;
   return CHANGE ;
```

[4] We name it a queue, but, as in AC-3, it has to be implemented as a set since in line 3 of Algorithm 1 and in line 5 of Algorithm 2 we add X_i to Q only if it does not already belong to it.

[5] In fact, McGregor's version of the function `Revise` differs from Mackworth's one. Algorithm `Revise3` is the Mackworth's version [Mackworth, 1977].

[6] The third parameter is useless for AC-3 but will be used in the next version of `Revise`.

[7] We take this name from [Van Hentenryck *et al.*, 1992], where Δ denotes the same thing, but for a different use.

Algorithm 3: `Subprocedures for AC2000`

```
function Propagation2000 (in Q: set): Boolean
    while Q ≠ ∅ do
        pick X_j from Q;
1       lazymode ← (|Δ(X_j)| < Ratio · |D(X_j)|);
        for each X_i such that C_ij ∈ C do
            if Revise2000(X_i, X_j, lazymode) then
                if D(X_i) = ∅ then return false ;
                Q ← Q ∪ {X_i};
2       reset Δ(X_j);
    return true ;

function Revise2000 (in X_i, X_j: variable;
                      in lazymode: Boolean ): Boolean
    CHANGE ← false;
    for each v_i ∈ D(X_i) do
3       if ¬lazymode or ∃v_j ∈ Δ(X_j)/C_ij(v_i, v_j) then
4           if ∄v_j ∈ D(X_j)/C_ij(v_i, v_j) then
                remove v_i from D(X_i);
                add v_i to Δ(X_i);
5               CHANGE ← true;
    return CHANGE ;
```

constraint checks. We save 44 constraint checks compared to AC-3.

Analysis

Let us first briefly prove `AC2000` correctness. Assuming AC-3 is correct, we just have to prove that the lazy mode does not let arc-inconsistent values in the domain. The only way the search for support for a value v_i in $D(X_i)$ is skipped, is when we could not find a support for v_i in $\Delta(X_j)$ (line 3 of `Revise2000`). Since $\Delta(X_j)$ contains all the values deleted from $D(X_j)$ since its last propagation (line 2 of `Propagation2000`), this means that v_i has exactly the same set of supports as before on C_{ij}. Thus, looking again for a support for v_i is useless. It remains consistent with C_{ij}.

The space complexity of `AC2000` is bounded above by the sizes of Q and Δ. Q is in $O(n)$ and Δ is in $O(nd)$, where d is the size of the largest domain. This gives a $O(nd)$ overall complexity. In this space complexity, it is assumed that we built `AC2000` with the variable-oriented propagation scheme, as recommended earlier. If we implement `AC2000` with the constraint-oriented propagation scheme of the original AC-3, we need to attach a $\Delta(X_j)$ to each constraint C_{ij} put in the queue. This implies a $O(ed)$ space complexity.

The organization of `AC2000` is the same as in AC-3. The main change is in function `Revise2000`, where $\Delta(X_j)$ and $D(X_j)$ are examined instead of only $D(X_j)$. Their total size is bounded above by d. This leads to a worst-case where d^2 checks are performed, as in `Revise3`. Thus, the overall time complexity is in $O(ed^3)$ since `Revise2000` can be called d times per constraint. This is as in AC-3.

4 AC2001

In Section 3, we proposed an algorithm, which, like AC-3, does not need special data structures to be maintained during search. (Except the current domains, which have to be maintained by any search algorithm performing some look-ahead.) During a call to `Revise2000`, for each v_i in $D(X_i)$, we have to look whether v_i has a support in $\Delta(X_j)$, to know whether it lost supports or not. In this last case, we have to look again for a support for v_i on C_{ij} in the whole $D(X_j)$ set. If we could remember what was the support found for v_i in $D(X_j)$ the last time we revised C_{ij}, the gain would be twofold: First, we would just have to check whether this last support has been removed from $D(X_j)$ or not, instead of exploring the set $\Delta(X_j)$. Second, when a support was effectively removed from $D(X_j)$, we would just have to explore the values in $D(X_j)$ that are "after" that last support since "predecessors" have already been checked before. Adding a very light extra data structure to remember the last support of a value on a constraint leads to the algorithm `AC2001` that we present in this section.

Let us store in $Last(X_i, v_i, X_j)$ the value that has been found as a support for v_i at the last call to `Revise2001`(X_i, X_j). The function `Revise2001` will always run in the lazy mode since the cost of checking whether the $Last$ support on C_{ij} of a value v_i has been removed from $D(X_j)$ is not dependent on the number of values removed from $D(X_j)$. A second change w.r.t. `AC2000` is that the structure Δ is no longer necessary since the test "$Last(X_i, v_i, X_j) \notin D(X_j)$" can replace the test "$Last(X_i, v_i, X_j) \in \Delta(X_j)$". The consequence is that `AC2001` can equally be used with a constraint-based or variable-based propagation. We only present the function `Revise2001`, which simply replaces the function `Revise3` in AC-3. The propagation procedure is that of AC-3, and the $Last$ structure has to be initialized to `NIL` at the beginning. In line 1 of `Revise2001` (see Algorithm 4) we check whether $Last(X_i, v_i, X_j)$ still belongs to $D(X_j)$. If it is not the case, we look for a new support for v_i in $D(X_j)$; otherwise nothing is done since $Last(X_i, v_i, X_j)$ is still in $D(X_j)$. In line 2 of `Revise2001` we can see the second advantage of storing $Last(X_i, v_i, X_j)$: If the supports are checked in $D(X_j)$ in a given ordering "$<_d$", we know that there isn't any support for v_i before $Last(X_i, v_i, X_j)$ in $D(X_j)$. Thus, we can look for a new support only on the values greater than $Last(X_i, v_i, X_j)$.

Algorithm 4: `Subprocedure for AC2001`

```
function Revise2001 (in X_i, X_j: variable): Boolean
    CHANGE ← false;
    for each v_i ∈ D(X_i) do
1       if Last(X_i, v_i, X_j) ∉ D(X_j) then
2           if ∃v_j ∈ D(X_j)/v_j >_d Last(X_i, v_i, X_j) ∧ C_ij(v_i, v_j)
            then Last(X_i, v_i, X_j) ← v_j;
            else
                remove v_i from D(X_i);
                add v_i to Δ(X_i);
                CHANGE ← true;
    return CHANGE ;
```

On the example of Section 3, when $(X_j, 11)$ is removed, `AC2001` simply checks for each $v_i \in [1..10]$ that $Last(X_i, v_i, X_j)$ still belongs to $D(X_j)$, and finds that $Last(X_i, 11, X_j)$ has been removed. Looking for a new sup-

		AC-3		AC2000		AC2001		*AC-6*$^{(*)}$
		#ccks	time	#ccks	time	#ccks	time	*time*
<150, 50, 500, 1250>	(under-constrained)	100,010	0.04	100,010	0.05	100,010	0.05	*0.07*
<150, 50, 500, 2350>	(over-constrained)	507,783	0.18	507,327	0.18	487,029	0.16	*0.10*
<150, 50, 500, 2296>	(phase transition)	2,860,542	1.06	1,601,732	0.69	688,606	0.34	*0.32*
<50, 50, 1225, 2188>	(phase transition)	4,925,403	1.78	3,038,280	1.25	1,147,084	0.61	*0.66*
SCEN#08	(arc inconsistent)	4,084,987	1.67	3,919,078	1.65	2,721,100	1.25	*0.51*

Table 1: Arc consistency results in mean number of constraint checks (#ccks) and mean cpu time in seconds (time) on a PC Pentium II 300MHz (50 instances generated for each random class). (*) The number of constraint checks performed by AC-6 is similar to that of AC2001, as discussed in Section 6.

port for 11 does not need any constraint check since $D(X_j)$ does not contain any value greater than $Last(X_i, 11, X_j)$, which was equal to 11. It saves 65 constraint checks compared to AC-3.

Analysis

Proving correctness of AC2001 can be done very quickly since the framework of the algorithm is very close to AC-3. They have exactly the same initialization phase except that AC2001 stores $Last(X_i, v_i, X_j)$, the support found for each v_i on each C_{ij}. (In line 1 it is assumed that NIL does not belong to $D(X_j)$.) During the propagation, they diverge in the way they revise an arc. As opposed to AC-3, Revise2001(X_i, X_j) goes into a search for support for a value v_i in $D(X_i)$ only if $Last(X_i, v_i, X_j)$ does not belong to $D(X_j)$. We see that checking that $Last(X_i, v_i, X_j)$ still belongs to $D(X_j)$ is sufficient to ensure that v_i still has a support in $D(X_j)$. And if a search for a new support has to be done, limiting this search to the values of $D(X_j)$ greater than $Last(X_i, v_i, X_j)$ w.r.t. to the ordering $<_d$ used to visit $D(X_j)$ is sufficient. Indeed, the previous call to Revise2001 stopped as soon as the value $Last(X_i, v_i, X_j)$ was found. It was then the smallest support for v_i in $D(X_j)$ w.r.t. $<_d$.

The space complexity of AC2001 is bounded above by the size of Q, and $Last$. Q is in $O(n)$ or $O(e)$, depending on the propagation scheme that is used (variable-based or constraint-based). $Last$ is in $O(ed)$ since each value v_i has a $Last$ pointer for each constraint involving X_i. This gives a $O(ed)$ overall complexity.

As in AC-3 and AC2000, the function Revise2001 can be called d times per constraint in AC2001. But, at each call to Revise2001(X_i, X_j), for each value $v_i \in D(X_i)$, there will be a test on the $Last(X_i, v_i, X_j)$, and a search for support only on the values of $D(X_j)$ greater than $Last(X_i, v_i, X_j)$. Thus, the total work that can be performed for a value v_i over the d possible calls to Revise2001 on a pair (X_i, X_j) is bounded above by d tests on $Last(X_i, v_i, X_j)$ and d constraint checks. The overall time complexity is then bounded above by $d \cdot (d+d) \cdot 2 \cdot e$, which is in $O(ed^2)$. This is optimal [Mohr and Henderson, 1986]. AC2001 is the first optimal arc consistency algorithm proposed in the literature that is free of any lists of supported values. Indeed, the other optimal algorithms, AC-4, AC-6, AC-7, and AC-Inference all use these lists.

5 Experiments

In the sections above, we presented two refinements of AC-3, namely AC2000 and AC2001. It remains to see whether they are effective in saving constraint checks and/or cpu time when compared to AC-3. AS we said previously, the goal is not to compete with AC-6/AC-7, which have very subtle data structure for the propagation phase. An improvement (even small) w.r.t. AC-3 would fulfill our expectations. However, we give AC-6 performances, just as a marker.

5.1 Arc consistency as a preprocessing

The first set of experiments we performed should permit to see the effect of our refinements when arc consistency is used as a preprocessing (without search). In this case, the chance to have some propagations is very small on real instances. We have to fall in the phase transition of arc consistency (see [Gent *et al.*, 1997]). So, we present results for randomly generated instances (those presented in [Bessière *et al.*, 1999]), and for only one real-world instance. For the random instances, we used a model B generator [Prosser, 1996]. The parameters are $< N, D, C/p1, T/p2 >$, where N is the number of variables, D the size of the domains, C the number of constraints (their density $p1 = 2C/N \cdot (N-1)$), and T the number of forbidden tuples (their tightness $p2 = T/D^2$). The real-world instance, SCEN#08, is taken from the FullRLFAP archive,[8] which contains instances of radio link frequency assignment problems (RLFAPs). They are described in [Cabon *et al.*, 1999]. The parameter Ratio used in AC2000 is set to 0.2. Table 1 presents the results for four classes of random instances plus the real-world one.

The upper two are under-constrained (<$150, 50, 500/0.045, 1250/$**0.5**>) and over-constrained (<$150, 50, 500/0.045, 2350/$**0.94**>) problems. They represent cases where there is little or no propagation to reach the arc consistent or arc inconsistent state. This is the best case for AC-3, which performs poorly during propagation. We can see that AC2000 and AC2001 do not suffer from this.

The third and fourth experiments are at the phase transition of arc consistency for sparse (<$150, 50, 500/$**0.045**$, 2296/0.918$>) and dense (<$50, 50, 1225/$**1.0**$, 2188/0.875$>) problems. We can assume there is much propagation on these problems before reaching the arc consistent state. This has a significant impact on the respective efficiencies of the algorithms. The smarter

[8] We thank the Centre d'Electronique de l'Armement (France).

	MAC-3		MAC2000		MAC2001		*MAC6*
	#ccks	time	#ccks	time	#ccks	time	*time*
SCEN#01	5,026,208	2.33	4,319,423	2.10	1,983,332	1.62	*2.05*
SCEN#11	77,885,671	39.50	77,431,840	38.22	9,369,298	21.96	*14.69*
GRAPH#09	6,269,218	2.95	5,164,692	2.57	2,127,598	1.99	*2.41*
GRAPH#10	6,790,702	3.04	5,711,923	2.65	2,430,109	1.85	*2.17*
GRAPH#14	5,503,326	2.53	4,253,845	2.13	1,840,886	1.66	*1.90*

Table 2: Results for search of the first solution with a MAC algorithm in mean number of constraint checks (#ccks) and mean cpu time in seconds (time) on a PC Pentium II 300MHz.

the algorithm is, the lower the number of constraint checks is. AC2001 dominates AC2000, which itself dominates AC-3. And the cpu time follows this trend.

The lower experiment reports the results for the SCEN#08. This is one of the instances in FullRLFAP for which arc consistency is sufficient to detect inconsistency.

5.2 Maintaining arc consistency during search

The second set of experiments we present in this section shows the effect of our refinements when arc consistency is maintained during search (MAC algorithm [Sabin and Freuder, 1994]) to find the first solution. We present results for all the instances contained in the FullRLFAP archive for which more than 2 seconds were necessary to find a solution or prove that none exists. We took again 0.2 for the Ratio in AC2000. It has to be noticed that the original question in these instances is not satisfiability but the search of the "best" solution, following some criteria. It is of course out of the scope of this paper.

From these instances we can see a slight gain for AC2000 on AC-3. On SCEN#11, it can be noticed that with a smaller Ratio, AC2000 slightly improves its performances. (Ratio = 0.05 seems to be the best.) A more significant gain can be seen for AC2001, with up to 9 times less constraint checks and twice less cpu time on SCEN#11. As for the experiments performed on random instances at the phase transition of arc consistency, this tends to show that the trick of storing the $Last$ data structure significantly pays off. However, we have to keep in mind that we are only comparing algorithms with simple data structures. This prevents them from reaching the efficiency of algorithms using lists of supported values when the amount of propagation is high, namely on hard problems. (E.g., a MAC algorithm using AC-6 for enforcing arc consistency needs only 14.69 seconds to solve the SCEN#11 instance.)

6 AC2001 vs AC-6

In the previous sections, we proposed two algorithms based on AC-3 to achieve arc consistency on a binary constraint network. AC2000 is close to AC-3, from which it inherits its $O(ed^3)$ time complexity and its $O(nd)$ space complexity. AC2001, thanks to its additional data structure, has an optimal $O(ed^2)$ worst-case time complexity, and an $O(ed)$ space complexity. These are the same characteristics as AC-6.[9] So, we can ask the question: "What are the differences between AC2001 and AC-6?".

Let us first briefly recall the AC-6 behavior [Bessière, 1994]. AC-6 looks for one support (the *first* one or *smallest* one with respect to the ordering $<_d$) for each value (X_i, v_i) on each constraint C_{ij} to prove that (X_i, v_i) is currently viable. When (X_j, v_j) is found as the smallest support for (X_i, v_i) on C_{ij}, (X_i, v_i) is added to $S[X_j, v_j]$, the list of values currently having (X_j, v_j) as smallest support. If (X_j, v_j) is removed from $D(X_j)$, it is added to the $DeletionSet$, which is the stream driving propagations in AC-6. When (X_j, v_j) is picked from the $DeletionSet$, AC-6 looks for the *next* support (i.e., greater than v_j) in $D(X_j)$ for each value (X_i, v_i) in $S[X_j, v_j]$. Notice that the $DeletionSet$ corresponds to $\sum_{X_j \in \mathcal{X}} \Delta(X_j)$ in AC2000, namely the set of values removed but not yet propagated.

To allow a closer comparison, we will suppose in the following that the $S[X_i, v_i]$ lists of AC-6 are split on each constraint C_{ij} involving X_i, leading to a structure $S[X_i, v_i, X_j]$, as in AC-7.

Property 1 *Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network. If we suppose AC2001 and AC-6 follow the same ordering of variables and values when looking for supports and propagating deletions, then, enforcing arc consistency on $\mathcal{P}$ with AC2001 requires the same constraint checks as with AC-6.*

Proof. Since they follow the same ordering, both algorithms perform the same constraint checks in the initialization phase: they stop search for support for a value v_i on C_{ij} as soon as the first v_j in $D(X_j)$ compatible with v_i is found, or when $D(X_j)$ is exhausted (then removing v_i). During the propagation phase, both algorithms look for a new support for a value v_i on C_{ij} only when v_i has lost its current support v_j in $D(X_j)$ (i.e., $v_i \in S[X_j, v_j, X_i]$ for AC-6, and $v_j = Last(X_i, v_i, X_j)$ for AC2001). Both algorithms start the search for a new support for v_i at the value in $D(X_j)$ immediately greater than v_j w.r.t. the $D(X_j)$ ordering. Thus, they will find the same new support for v_i on C_{ij}, or will remove v_i, at the same time, and with the same constraint checks. And so on. □

From property 1, we see that the difference between AC2001 and AC-6 cannot be characterized by the number of constraint checks they perform. We will then focus on the way they find which values should look for a new support. For that, both algorithms handle their specific data structures. Let us characterize the number of times each of them checks

[9] We do not speak about AC-7 here, since it is the only one among these algorithms to deal with the bidirectionality of the constraints (namely, the fact that $C_{ij}(v_i, v_j) = C_{ji}(v_j, v_i)$).

its own data structure when a set $\Delta(X_j)$ of deletions is propagated on a given constraint C_{ij}.

Property 2 *Let C_{ij} be a constraint in a network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. Let $\Delta(X_j)$ be a set of values removed from $D(X_j)$ that have to be propagated on C_{ij}. If,*
$A = |\Delta(X_j)| + \sum_{v_j \in \Delta(X_j)} |S[X_j, v_j, X_i]|$,
$B = |D(X_i)|$, *and*
C = *# checks performed on C_{ij} to propagate $\Delta(X_j)$,*
then, $A + C$ and $B + C$ represent the number of operations AC-6 and `AC2001` *will respectively perform to propagate $\Delta(X_j)$ on C_{ij}.*

Proof. From property 1 we know that AC-6 and `AC2001` perform the same constraint checks. The difference is in the process leading to them. AC-6 traverses the $S[X_j, v_j, X_i]$ list for each $v_j \in \Delta(X_j)$ (i.e., A operations), and `AC2001` checks whether $Last(X_i, v_i, X_j)$ belongs to $D(X_j)$ for every v_i in $D(X_i)$ (i.e., B operations). □

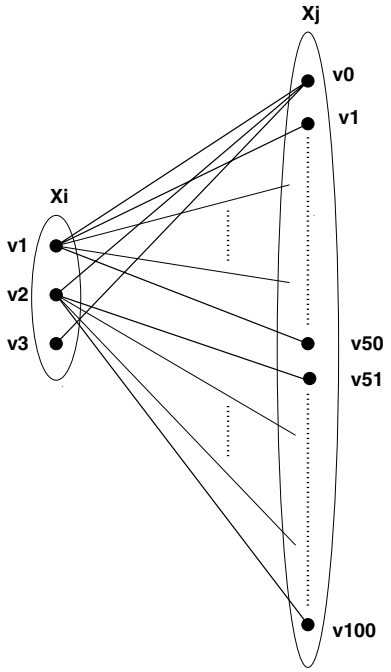

Figure 1: The constraint example

We illustrate this on the extreme case presented in Figure 1. In that example, the three values of X_i are all compatible with the first value v_0 of X_j. In addition, (X_i, v_1) is compatible with all the values of X_j from v_1 to v_{50}, and (X_i, v_2) with all the values of X_j from v_{51} to v_{100}. Imagine that for some reason, the value v_3 has been removed from $D(X_i)$ (i.e., $\Delta(X_i) = \{v_3\}$). This leads to $A = 1$, $B = 101$, and $C = 0$, which is a case in which propagating with AC-6 is much better than with `AC2001`, even if none of them needs any constraint check. Indeed, AC-6 just checks that $S[X_i, v_3, X_j]$ is empty,[10] and stops. `AC2001` takes one by one the 101 values v_j of $D(X_j)$ to check that their $Last(X_j, v_j, X_i)$ is not in $\Delta(X_i)$. Imagine now that the values v_1 to v_{100} of $D(X_j)$ have been removed (i.e., $\Delta(X_j) = \{v_1, \ldots, v_{100}\}$). Now, $A = 100$, $B = 3$, and $C = 0$. This means that `AC2001` will clearly outperform AC-6. Indeed, AC-6 will check for all the 100 values v_j in $\Delta(X_j)$ that $S[X_j, v_j, X_i]$ is empty,[11] while `AC2001` just checks that $Last(X_i, v_i, X_j)$ is not in $\Delta(X_j)$ for the 3 values in $D(X_i)$.

[10] The only value compatible with (X_i, v_3) is (X_j, v_0), which is currently supported by (X_i, v_1).

Discussion

Thanks to property 2, we have characterized the amount of effort necessary to AC-6 and `AC2001` to propagate a set $\Delta(X_j)$ of removed values on a constraint C_{ij}. $A - B$ gives us information on which algorithm is the best to propagate $\Delta(X_j)$ on C_{ij}. We can then easily imagine a new algorithm, which would start with an AC-6 behavior on all the constraints, and would switch to `AC2001` on a constraint C_{ij} as soon as $A - B$ would be positive on this constraint (and then forget the S lists on C_{ij}). Switching from `AC2001` to AC-6 is no longer possible on this constraint because we can deduce in constant time that $Last(X_i, v_i, X_j) = v_j$ when v_i belongs to $S[X_j, v_j, X_i]$, but we cannot obtain cheaply $S[X_j, v_j, X_i]$ from the $Last$ structure. A more elaborated version would maintain the S lists even in the `AC2001` behavior (putting v_i in $S[X_j, v_j, X_i]$ each time v_j is found as being the $Last(X_i, v_i, X_j)$). This would permit to switch from AC-6 to `AC2001` or the reverse at any time on any constraint in the process of achieving arc consistency. These algorithms are of course far from our initial purpose of emphasizing easiness of implementation since they require the S and $Last$ structures to be maintained during search.

7 Non-binary versions

Both `AC2000` and `AC2001` can be extended to deal with non-binary constraints. A support is now a tuple instead of a value. Tuples in a constraint $C(X_{i_1}, \ldots, X_{i_q})$ are ordered w.r.t. the ordering $<_d$ of the domains, combined with the ordering of $X_{i_1}, \ldots, X_{i_q}$ (or any order used when searching for support). Once this ordering is defined, a call to `Revise2000`(X_i, C) —because of a set $\Delta(X_j)$ of values removed from $D(X_j)$— simply checks for each $v_i \in D(X_i)$ whether there exists a support τ of v_i on the constraint C for which $\tau[X_j]$ (the value of X_j in the tuple) belongs to $\Delta(X_j)$. If yes, it looks for a new support for v_i on C. `Revise2001`(X_i, C) checks for each $v_i \in D(X_i)$ whether $Last(X_i, v_i, C)$, which is a tuple, still belongs to $D(X_{i_1}) \times \cdots \times D(X_{i_q})$ before looking for a new support for v_i on C.

This extension to non-binary constraints is very simple to implement. However, it has to be handled with care when the variable-oriented propagation is used, as recommended for `AC2000`. (With a constraint-based propagation, a $\Delta(X_j)$ set is duplicated for each constraint put in the queue to propagate it.) Variable-based propagation is indeed less precise in

[11] Indeed, (X_j, v_0) is the current support for the three values in $D(X_i)$ since it is the smallest in $D(X_j)$ and it is compatible with every value in $D(X_i)$.

the way it drives propagation than constraint-based propagation. Take the constraint $C(X_{i_1}, X_{i_2}, X_{i_3})$ as an example. If $D(X_{i_1})$ and $D(X_{i_2})$ are modified consecutively, X_{i_1} and X_{i_2} are put in the queue Q consecutively. Picking X_{i_1} from Q implies the calls to $\texttt{Revise}(X_{i_2}, C)$ and $\texttt{Revise}(X_{i_3}, C)$, and picking X_{i_2} implies the calls to $\texttt{Revise}(X_{i_1}, C)$ and $\texttt{Revise}(X_{i_3}, C)$. We see that $\texttt{Revise}(X_{i_3}, C)$ is called twice while once was enough. To overcome this weakness, we need to be more precise in the way we propagate deletions. The solution, while being technically simple, is more or less dependent on the architecture of the solver in which it is used. Standard techniques are described in [ILOG, 1998; Laburthe, 2000].

8 Conclusion

We presented `AC2000` and `AC2001`, two refinements in AC-3. The first one improves slightly AC-3 in efficiency (number of constraint checks and cpu time) although it does not need any new data structure to be maintained during search. The second, `AC2001`, needs an additional data structure, the $Last$ supports, which should be maintained during search. This data structure permits a significant improvement on AC-3, and decreases the worst-case time complexity to the optimal $O(ed^2)$. `AC2001` is the first algorithm in the literature achieving optimally arc consistency while being free of any lists of supported values. Its behavior is compared to that of AC-6, making a contribution to the understanding of the different AC algorithms, and opening an opportunity of improvement. This is in the same vein as the work on AC-3 vs AC-4 [Wallace, 1993], which was leading up to AC-6.

Acknowledgements

We would like to thank Philippe Charman who pointed out to us the negative side of value-oriented propagation. The first author also wants to thank all the members of the OCRE team for the discussions we had about the specification of the CHOCO language.

References

[Bessière *et al.*, 1995] C. Bessière, E. C. Freuder, and J. C. Régin. Using inference to reduce arc consistency computation. In *Proceedings IJCAI'95*, pages 592–598, Montréal, Canada, 1995.

[Bessière *et al.*, 1999] C. Bessière, E.C. Freuder, and J.C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

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

[Cabon *et al.*, 1999] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4:79–89, 1999.

[Chmeiss and Jégou, 1998] A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.

[Gent *et al.*, 1997] I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proceedings CP'97*, pages 327–340, Linz, Austria, 1997.

[ILOG, 1998] ILOG. *User's manual*. ILOG Solver, 4.3 edition, 1998.

[Laburthe, 2000] F. Laburthe. *User's manual*. CHOCO, 0.39 edition, 2000.

[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 isomorphism. *Information Science*, 19:229–250, 1979.

[Mohr and Henderson, 1986] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Prosser, 1996] P. Prosser. An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[Sabin and Freuder, 1994] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP'94*, Seattle WA, 1994.

[Van Hentenryck *et al.*, 1992] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Wallace, 1993] R.J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proceedings IJCAI'93*, pages 239–245, Chambéry, France, 1993.

Making AC-3 an Optimal Algorithm

Yuanlin Zhang and **Roland H.C. Yap***
School of Computing, National University of Singapore
Lower Kent Ridge Road, 119260, Singapore
{zhangyl, ryap}@comp.nus.edu.sg

Abstract

The AC-3 algorithm is a basic and widely used arc consistency enforcing algorithm in Constraint Satisfaction Problems (CSP). Its strength lies in that it is simple, empirically efficient and extensible. However its worst case time complexity was not considered optimal since the first complexity result for AC-3 [Mackworth and Freuder, 1985] with the bound $\mathcal{O}(ed^3)$, where e is the number of constraints and d the size of the largest domain. In this paper, we show suprisingly that AC-3 achieves the optimal worst case time complexity with $\mathcal{O}(ed^2)$. The result is applied to obtain a path consistency algorithm which has the same time and space complexity as the best known theoretical results. Our experimental results show that the new approach to AC-3 is comparable to the traditional AC-3 implementation for simpler problems where AC-3 is more efficient than other algorithms and significantly faster on hard instances.

1 Introduction

Arc consistency is a basic technique for solving Constraint Satisfaction Problems (CSP) and variations of arc consistency are used in many AI and constraint applications. There have been many algorithms developed for arc consistency such as AC-3 [Mackworth, 1977], AC-4 [Mohr and Henderson, 1986], AC-6 [Bessiere, 1994] and AC-7 [Bessiere *et al.*, 1999]. The AC-3 algorithm was proposed in 1977 [Mackworth, 1977]. The first worst case analysis of AC-3 [Mackworth and Freuder, 1985] gives a complexity of $\mathcal{O}(ed^3)$, where e is the number of constraints and d the size of largest domain. This result is deeply rooted in the CSP literature (eg. [Wallace, 1993; Bessiere *et al.*, 1999]) and thus AC-3 is typically considered to be non-optimal. Other algorithms such as AC-4, AC-6, AC-7 are considered theoretically optimal, with time complexity $\mathcal{O}(ed^2)$. As far as we are aware, there has not been any effort to improve the theoretical bound of AC-3 to be optimal. Here, we re-examine AC-3 for a number of reasons. Firstly, AC-3 is one of the simplest AC algorithms and is known to be practically efficient [Wallace, 1993]. The simplicity of arc revision in AC-3 makes it convenient for implementation and amenable to various extensions for many constraint systems. Thus while AC-3 is considered as being sub-optimal, it often is the algorithm of choice and can outperform other theoretically optimal algorithms.

In this paper, we show that AC-3 achieves worst case optimal time complexity of $\mathcal{O}(ed^2)$. This result is surprising since AC-3 being a coarse grained "arc revision" algorithm [Mackworth, 1977], is considered to be non-optimal. The known results for optimal algorithms are all on fine grained "value revision" algorithms. Preliminary experiments show that the new AC-3 is comparable to the traditional implementations on easy CSP instances where AC-3 is known to be substantially better than the optimal fine grained algorithms. In the hard problem instances such as those from the phase transition, the new AC-3 is significantly better and is comparable to the best known algorithms such as AC-6. We also show that the results for AC-3 can be applied immediately to obtain a path consistency algorithm which has the same time and space complexity as the best known theoretical results. [1]

2 Background

In this section we give some background material and notation used herein. The definitions for general CSP follow [Montanari, 1974; Mackworth, 1977].

Definition 1 *A* Constraint Satisfaction Problem *(N, D, C) consists of a finite set of variables $N = \{1, \cdots, n\}$, a set of domains $D = \{D_1, \cdots, D_n\}$, where $i \in D_i$, and a set of constraints $C = \{c_{ij} \mid i, j \in N\}$, where each constraint c_{ij} is a binary relation between variables i and j. For the problem of interest here, we require that $\forall x, y \;\; x \in D_i, y \in D_j, (x, y) \in c_{ij}$ if and only if $(y, x) \in c_{ji}$.*

For simplicity, in the above definition we consider only binary constraints, omitting the unary constraint on any variable [Mackworth, 1977]. Without loss of generality we assume there is only one constraint between each pair of variables.

Definition 2 *The* constraint graph *of a CSP (N, D, C) is the graph $G = (V, E)$ where $V = N$ and $E = \{(i, j) \mid \exists c_{ij} \in C$ or $\exists c_{ji} \in C\}$.*

*The Logistics Institute Asia-Pacific

[1] A related paper by Bessiere and Regin appears in this proceedings.

The arcs in CSP refer to the directed edges in G. Throughout this paper, n denotes the number of variables, d the size of the largest domain, and e the number of constraints in C.

Definition 3 *Given a CSP (N, D, C), an arc (i, j) of its constraint graph is* arc consistent *if and only if $\forall x \in D_i$, there exists $y \in D_j$ such that $c_{ij}(x, y)$ holds. A CSP (N, D, C) is arc consistent if and only if each arc in its constraint graph is arc consistent.*

The AC-3 algorithm for enforcing arc consistency on a CSP is given in figure 2. The presentation follows [Mackworth, 1977; Mackworth and Freuder, 1985] with a slight change in notation and node consistency removed.

```
procedure REVISE((i, j))
  begin
    DELETE ← false
    for each x ∈ D_i do
1.    if there is no y ∈ D_j such that c_ij(x, y) then
          delete x from D_i;
          DELETE ← true
      endif
    return DELETE
  end
```

Figure 1: procedure REVISE for AC-3

```
algorithm AC-3
  begin
1.  Q ← {(i, j) | c_ij ∈ C or c_ji ∈ C, i ≠ j}
    while Q not empty do
      select and delete any arc (k, m) from Q;
2.    if REVISE((k, m)) then
3.      Q ← Q ∪ {(i, k) | (i, k) ∈ C, i ≠ k, i ≠ m}
    endwhile
  end
```

Figure 2: The AC-3 algorithm

The task of REVISE$((i, j))$ in Fig 1 is to remove those invalid values not related to any other value with respect to arc (i, j). We will show in section 3 that different implementations of line 1 lead to different worst case complexities. As such, we argue that it is more useful to think of AC-3 as a framework rather than a specific algorithm. In AC-3, a CSP is modeled by its constraint graph G, and what AC-3 does is to revise all arcs $(*, i) = \{(k, i) | (k, i) \in E(G)\}$ (line 3 in Fig 2) except some special arc if the domain of variable i is modified by REVISE$((i, j))$. A queue Q is used to hold all arcs that need to be revised.The traditional understanding of AC-3 is given by the following theorem whose proof from [Mackworth and Freuder, 1985] is modified in order to facilitate the presentation in Section 3.

Theorem 1 *[Mackworth and Freuder, 1985] Given a CSP (N, D, C), the time complexity of AC-3 is $\mathcal{O}(ed^3)$.*

Proof Consider the times of revision of each arc (i, j). (i, j) is revised if and only if it enters Q. The observation is that arc (i, j) enters Q if and only if some value of j is deleted (line 2–3 in fig 2). So, arc (i, j) enters Q at most d times and thus is revised d times. Given that the number of arcs are $2e$, REVISE(i, j) is executed $\mathcal{O}(ed)$ times. The complexity of REVISE$((i, j))$ in Fig 1 is at most d^2. □

The reader is referred to [Mackworth, 1977; Mackworth and Freuder, 1985] for more details and motivations concerning arc consistency.

3 A New View of AC-3

The traditional view of AC-3 with the worst case time complexity of $\mathcal{O}(ed^3)$ (described by theorem 1) is based on a naive implementation of line 1 in Fig 1 that y is always searched from scratch. Hereafter, for ease of presentation, we call the classical implementation AC-3.0. The new approach to AC-3 in this paper, called AC-3.1, is based on the observation that y in line 1 of Fig 1 needn't be searched from scratch even though the same arc (i, j) may enter Q many times, The search is simply resumed from the point where it stopped in the previous revision of (i, j). This idea is implemented by procedure EXIST$y((i, x), j)$ in Fig 3.

Assume without loss of generality that each domain D_i is associated with a total ordering. ResumePoint$((i, x), j)$ remembers the first value $y \in D_j$ such that $c_{ij}(x, y)$ holds in the previous revision of (i, j). The *succ*(y, D_j^0) function, where D_j^0 denotes the domain of j before arc consistency enforcing, returns the successor of y in the ordering of D_j^0 or NIL, if no such element exists. NIL is a value not belonging to any domain and precedes all values in any domain.

```
procedure EXISTy((i, x), j)
  begin
    y ← ResumePoint((i, x), j);
1:  if y ∈ D_j then % y is still in the domain
      return true;
    else
2:      while ((y ← succ(y, D_j^0) and (y ≠ NIL))
          if y ∈ D_j and c_ij(x, y) then
            ResumePoint((i, x), j) ← y;
            return true
          endif;
        return false
    endif
  end
```

Figure 3: Procedure for searching y in REVISE(i, j)

Theorem 2 *The worst case time complexity of AC-3 can be achieved in O(ed^2).*

Proof Here it is helpful to regard the execution of AC-3.1 on a CSP instance as a sequence of calls to EXIST$y((i, x), j)$. Consider the time spent on $x \in D_i$ with respect to (i, j). As in theorem 1, an arc (i, j) enters Q at most d times. So, with respect to (i, j), any value $x \in D_i$ will be passed to EXIST$y((i, x), j)$ at most d times . Let the complexity of each execution of EXIST$y((i, x), j)$ be t_l $(1 \leq l \leq d)$. t_l can be considered as 1 if $y \in D_j$ (see line 1 in fig 3) and

otherwise it is s_l which is simply the number of elements in D_j skipped before next y is found (the while loop in line 2). Furthermore, the total time spent on $x \in D_i$ with respect to (i, j) is $\sum_1^d t_l \leq \sum_1^d 1 + \sum_1^d s_l$ where $s_l = 0$ if $t_l = 1$. Observe that in EXIST$y((i, x), j)$ the while loop (line 2) will skip an element in D_j at most once with respect to $x \in D_i$. Therefore, $\sum_1^d s_l \leq d$. This gives, $\sum_1^d t_l \leq 2d$. For each arc (i, j), we have to check at most d values in D_i and thus at most O(d^2) time will be spent on checking arc (i, j). Thus, the complexity of the new implementation of AC-3 is O(ed^2) because the number of arcs in constraint graph of the CSP is $2e$. □

The space complexity of AC-3.1 is not as good as the traditional implementation of AC-3. AC-3.1 needs additional space to remember the resumption point of any value with respect to any related constraint. It can be shown that the extra space required is O(ed), which is the same as AC-6.

The same idea behind AC-3.1 applies to path consistency enforcing algorithms. If one pair $(x, y) \in c_{kj}$ is removed, we need to recheck all pairs $(x, *) \in c_{ij}$ with respect to $c_{kj} \circ c_{ik}$ (the composition of c_{ik} and c_{kj}), and $(*, y) \in c_{lk}$ with respect to $c_{jk} \circ c_{lj}$. The resumption point $z \in D_k$ is remembered for any pair (x, y) of any constraint c_{ij} with respect to any intermediate variable k such that $c_{ik}(x, z), c_{kj}(z, y)$ both hold. ResumePoint$((i, x), (j, y), k)$ is employed to achieve the above idea in the algorithm in fig 4 which is partially motivated by the algorithm in [Chmeiss and Jegou, 1996].

```
algorithm PC
  begin
    INITIALIZE(Q);
    while Q not empty do
        Select and delete any ((i,x),j) from Q;
        REVISE_PC((i,x),j,Q))
    endwhile
  end
procedure INITIALIZE(Q)
  begin
    for any i,j,k ∈ N do
      for any x ∈ D_i, y ∈ D_j such that c_ij(x,y) do
        if there is no z ∈ D_k such that c_ik(x,z) ∧ c_kj(z,y)
        then
            c_ij(x,y) ← false;
            c_ji(y,x) ← false;
            Q ← Q ∪ {(i,x),j} ∪ {(j,y),i}
        else ResumePoint((i,x),(j,y),k) ← z
  end
```

Figure 4: Algorithm of Path Consistency Enforcing

By using a similar analysis to the proof of theorem 2, we have the following result.

Theorem 3 *The time complexity of the algorithm PC is O(n^3d^3) with space complexity O(n^3d^2).*

The time complexity and space complexity of the PC algorithm here are the same as the best known theoretical results [Singh, 1996].

```
procedure REVISE_PC( (i,x),k,Q)
  begin
    for any j ∈ N, k ≠ i, k ≠ j do
      for any y ∈ D_j such that c_ij(x,y) do
        z ← ResumePoint((i,x),(j,y),k);
        while not ((z ≠ NIL) ∧ c_ik(x,z) ∧ c_kj(z,y)
          do   z ← succ(z, D_k^0);
        if not ((c_ik(x,z) ∧ c_kj(z,y)) then
          Q ← Q ∪ {((i,x),j} ∪ {((j,y),i)}
        else ResumePoint((i,x),(j,y),k)) ← z
      endfor
  end
```

Figure 5: Revision procedure for PC algorithm

4 Preliminary experimental results

In this paper, we present some preliminary experimental results on the efficiency of AC-3. While arc consistency can be applied in the context of search (such as [Bessiere and Regin, 1996]), we focus on the performance statistics of the arc consistency algorithms alone.

The experiments are designed to compare the empirical performance of the new AC-3.1 algorithm with both the classical AC-3.0 algorithm and a state-of-the-art algorithm on a range of CSP instances with different properties.

There have been many experimental studies on the performance of general arc consistency algorithms [Wallace, 1993; Bessiere, 1994; Bessiere *et al.*, 1999]. Here, we adopt the choice of problems used in [Bessiere *et al.*, 1999], namely some random CSPs, Radio Link Frequency Assignment problems (RLFAPs) and the Zebra problem. The zebra problem is discarded as it is too small for benchmarking. Given the experimental results of [Bessiere *et al.*, 1999], AC-6 is chosen as a representative of a state-of-the-art algorithm because of its good timing performance over the problems of concern. In addition, an artificial problem DOMINO is designed to study the worst case performance of AC-3.

Randomly generated problems: As in [Frost *et al.*, 1996], a random CSP instance is characterized by n, d, e and the tightness of each constraint. The *tightness of a constraint* c_{ij} is defined to be $|D_i \times D_j| - |c_{ij}|$, the number of pairs NOT permitted by c_{ij}. A randomly generated CSP in our experiments is represented by a tuple (n, d, e, *tightness*). We use the first 50 instances of each of the following random problems generated using the initial seed 1964 (as in [Bessiere *et al.*, 1999]): (i) $P1$: under constrained CSPs $(150, 50, 500, 1250)$ where all generated instances are already arc consistent; (ii) $P2$: over constrained CSPs $(150, 50, 500, 2350)$ where all generated instances are *inconsistent* in the sense that some domain becomes empty in the process of arc consistency enforcing; and (iii) problems in the phase transition [Gent *et al.*, 1997] $P3$: $(150, 50, 500, 2296)$ and $P4$: $(50, 50, 1225, 2188)$. The $P3$ and $P4$ problems are further separated into the arc consistent instances, labeled as *ac*, which can be made arc consistent at the end of arc consistency enforcing; and inconsistent instances labeled as *inc*. More details on the choices for P1 to P4 can be found in [Bessiere *et al.*, 1999].

RLFAP: The RLFAP [Cabon *et al.*, 1999] is to assign frequencies to communication links to avoid interference. We use the CELAR instances of RLFAP which are real-life problems available at `ftp://ftp.cs.unh.edu/pub/csp/archive/code/benchmarks`.

DOMINO: Informally the DOMINO problem is an undirected constraint graph with a cycle and a trigger constraint. The domains are $D_i = \{1, 2, \cdots, d\}$. The constraints are $C = \{c_{i(i+1)} \mid i < n\} \cup \{c_{1n}\}$ where $c_{1n} = \{(d, d)\} \cup \{(x, x+1) \mid x < d\}$ is called the *trigger constraint* and the other constraints in C are identity relations. A DOMINO problem instance is characterized by two parameters n and d. The trigger constraint will make one value invalid during arc consistency and that value will trigger the domino effect on the values of all domains until each domain has only one value d left. So, each revision of an arc in AC-3 algorithms can only remove one value while AC-6 only does the necessary work. This problem is used to illustrate the differences between AC-3 like algorithms and AC-6. The results explain why arc revision oriented algorithms may not be so bad in the worst case as one might imagine.

		AC-3.0	AC-3.1	AC-6
P1	#ccks	100,010	100,010	100,010
	time(50)	0.65	0.65	1.13
P2	#ccks	494,079	475,443	473,694
	time(50)	1.11	1.12	1.37
P3(ac)	#ccks	2,272,234	787,151	635,671
	time(25)	2.73	1.14	1.18
P3(inc)	#ccks	3,428,680	999,708	744,929
	time(25)	4.31	1.67	1.69
P4(ac)	#ccks	3,427,438	1,327,849	1,022,399
	time(21)	3.75	1.70	1.86
P4(inc)	#ccks	5,970,391	1,842,210	1,236,585
	time(29)	8.99	3.63	3.54

Table 1: Randomly generated problems

RFLAP		AC-3.0	AC-3.1	AC-6
#3	#ccks	615,371	615,371	615,371
	time(20)	1.47	1.70	2.46
#5	#ccks	1,762,565	1,519,017	1,248,801
	time(20)	4.27	3.40	5.61
#8	#ccks	3,575,903	2,920,174	2,685,128
	time(20)	8.11	6.42	8.67
#11	#ccks	971,893	971,893	971,893
	time(20)	2.26	2.55	3.44

Table 2: CELAR RLFAPs

Some details of our implementation of AC-3.1 and AC-3.0 are as follows. We implement domain and related operations by employing a double-linked list. The Q in AC-3 is implemented as a queue of nodes on which arcs incident will be revised [Chmeiss and Jegou, 1996] . A new node will be put at the end of the queue. Constraints in the queue are revised in a FIFO order. The code is written in C++ with *g++*. The experiments are run on a Pentium III 600 processor with Linux.

d		AC-3.0	AC-3.1	AC-6
100	#ccks	17,412,550	1,242,550	747,551
	time(10)	5.94	0.54	0.37
200	#ccks	136,325,150	4,985,150	2,995,151
	time(10)	43.65	2.21	1.17
300	#ccks	456,737,750	11,227,750	6,742,751
	time(10)	142.38	5.52	2.69

Table 3: DOMINO problems

For AC-6, we note that in our experiments, using a single currently supported list of a values is faster than using multiple lists with respect to related constraints proposed in [Bessiere *et al.*, 1999]. This may be one reason why AC-7 is slower than AC-6 in [Bessiere *et al.*, 1999]. Our implementation of AC-6 adopts a single currently supported list.

The performance of arc consistency algorithms here is measured along two dimensions: running time and number of constraint checks (#ccks). A *raw constraint check* tests if a pair (x, y) where $x \in D_i$ and $y \in D_j$ satisfies constraint c_{ij}. In this experiment we assume constraint check is cheap and thus the raw constraint and additional checks (e.g. line 1 in Figs 3) in both AC-3.1 and AC-6 are counted. In the tabulated experiment results, *#ccks* represents the average number of checks on tested instances, and *time(x)* the time in seconds on x instances.

The results for randomly generated problems are listed in Table 1. For the under constrained problems $P1$, AC-3.1 and AC-3.0 have similar running time. No particular slowdown for AC-3.1 is observed. In the over constrained problems $P2$, the performance of AC-3.1 is close to AC-3.0 but some constraint checks are saved. In the hard phase transition problems $P3$ and $P4$, AC-3.1 shows significant improvement in terms of both the number of constraint checks and the running time, and is better than or close to AC-6 in timing.

The results for CELAR RLFAP are given in Table 2. In simple problems, RLFAP#3 and RLFAP#11, which are already arc consistent before the execution of any AC algorithm, no significant slowdown of AC-3.1 over AC-3.0 is observed. For RLFAP#5 and RLFAP#8, AC-3.1 is faster than both AC-3.0 and AC-6 in terms of timing.

The reason why AC-6 takes more time while making less checks can be explained as follows. The main contribution to the slowdown of AC-6 is the maintenance of the currently supported list of each value of all domains. In order to achieve space complexity of $\mathcal{O}(ed)$, when a value in the currently supported list is checked, the space occupied in the list by that value has to be released. Our experiment shows that the overhead of maintaining the list doesn't compensate for the savings from less checks under the assumption that constraint checking is cheap.

The DOMINO problem is designed to show the gap between AC-3 implementations and AC-6. Results in Table 3 show that AC-3.1 is about half the speed of AC-6. This can

be explained by a variation of the proof in section 3, in AC-3.1 the time spent on justifying the validity of a value with respect to a constraint is at most $2d$ while in AC-6 it is at most d. The DOMINO problem also shows that AC-3.0 is at least an order of magnitude slower in time with more constraint checks over AC-3.1 and AC-6.

In summary, our experiments on randomly generated problems and RLFAPs show the new approach to AC-3 has a satisfactory performance on both simple problems and hard problems compared with the traditional view of AC-3 and state-of-the-art algorithms.

5 Related work and discussion

Some related work is the development of general purpose arc consistency algorithms AC-3, AC-4, AC-6 , AC-7 and the work of [Wallace, 1993]. We summarize previous algorithms before discussing how this paper gives an insight into AC-3 as compared with the other algorithms.

An arc consistency algorithm can be classified by its method of propagation. So far, two approaches are employed in known efficient algorithms: arc oriented and value oriented. Arc oriented propagation originates from AC-1 and its underlying computation model is the constraint graph. Value oriented propagation originates from AC-4 and its underlying computation model is the value based constraint graph.

Definition 4 *The* value based constraint graph *of a CSP(N, D, C) is G=(V, E) where* $V = \{i.x \mid i \in N, x \in D_i\}$ *and* $E = \{\{i.x, j.y\} \mid x \in D_i, y \in D_j, c_{ij} \in C\}$.

Thus a more rigorous name for the traditional constraint graph may be *variable based constraint graph*. The key idea of value oriented propagation is that once a value is removed only those values related to it will be checked. Thus it is more fine grained than arc oriented propagation. Algorithms working with variable and value based constraint graph are also called *coarse grained algorithms* and *fine grained algorithms* respectively. An immediate observation is that compared with variable based constraint graph, time complexity analysis in value based constraint graph is straightforward.

Given a computation model of propagation, the algorithms differ in the implementation details. For variable based constraint graph, AC-3 [Mackworth, 1977] is an "open implementation". The approach in [Mackworth and Freuder, 1985] can be regarded as a realized implementation. The new view of AC-3 presented in this paper can be thought of as another implementation with optimal worst case complexity. Our new approach simply remembers the result obtained in previous revision of an arc while in the old one, the choice is to be lazy, forgetting previous computation. Other approaches to improving the space complexity of this model is [Chmeiss and Jegou, 1996]. For value based constraint graph, AC-4 is the first implementation and AC-6 is a lazy version of AC-4. AC-7 is based on AC-6 and it exploits the *bidirectional property* that given c_{ij}, c_{ji} and $x \in D_i, y \in D_j$, $c_{ij}(x, y)$ if and only if $c_{ji}(y, x)$.

Another aspect is the general properties or knowledge of a CSP which can be isolated from a specific arc consistency enforcing algorithm. Examples are AC-7 and AC-inference. We note that the idea of *metaknowledge* [Bessiere *et al.*, 1999] can be applied to algorithms of both computing models. For example, in terms of the number of *raw* constraint checks, the bidirectionality can be employed in coarse grained algorithm, eg. in [Gaschnig, 1978], however it may not be fully exploited under the variable based constraint graph model. Other propagation heuristics [Wallace, 1992] such as propagating deletion first [Bessiere *et al.*, 1999] are also applicable to algorithms of both models. This is another reason why we did not include AC-7 in our experimental comparison.

We have now a clear picture on the relationship between the new approach to AC-3 and other algorithms. AC-3.1 and AC-6 are methodologically different. From a technical perspective, the time complexity analysis of the new AC-3 is different from that of AC-6 where the worst case time complexity analysis is straightforward. The point of commonality between the new AC-3 and AC-6 is that they face the same problem: the domain may shrink in the process of arc consistency enforcing and thus the recorded information may not be always *correct*. This makes some portions of the new implementation of the AC-3.1 similar to AC-6. We remark that the proof technique in the traditional view of AC-3 does not directly lead to the new AC-3 and its complexity results.

The number of raw constraint checks is also used to evaluate practical efficiency of CSP algorithms. In theory, applying bidirectionality to all algorithms will result in a decrease of raw constraint checks. However, if the cost of raw constraint check is cheap, the overhead of using bidirectionality may not be compensated by its savings as demonstrated by [Bessiere *et al.*, 1999].

It can also be shown that if the same ordering of variables and values are processed, AC-3.1 and the classical AC-6 have the same number of raw constraint checks. AC-3.0 and AC-4 will make no less *raw* constraint checks than AC-3.1 and AC-6 respectively.

AC-4 does not perform well in practice [Wallace, 1993; Bessiere *et al.*, 1999] because it *reaches* the worst case complexity both theoretically and in actual problem instances when constructing the value based constraint graph for the instance. Other algorithms like AC-3 and AC-6 can take advantage of some instances being simpler where the worst case doesn't occur. In practice, both artificial and real life problems rarely make algorithms behave in the worst case except for AC-4. However, the value based constraint graph induced from AC-4 provides a convenient and accurate tool for studying arc consistency.

Given that both variable and value based constraint graph can lead to worst case optimal algorithms, we consider their strength on some special constraints: functional, monotonic and anti-functional. For more details, see [Van Hentenryck *et al.*, 1992] and [Zhang and Yap, 2000].

For coarse grained algorithms, it can be shown that for *monotonic* and *anti-monotonic* constraints arc consistency can be done with complexity of $\mathcal{O}(ed)$ (eg. using our new view of AC-3). With fine grained algorithms, both AC-4 and AC-6 can deal with *functional* constraints. We remark that the particular distance constraints in RLFAP can be enforced to be arc consistent in $\mathcal{O}(ed)$ by using a coarse grained algorithm. It is difficult for coarse grained algorithm to deal with functional constraints and tricky for fine grained algorithm to

monotonic constraints.

In summary, there are coarse grained and fine grained algorithms which are competitive given their optimal worst case complexity and good empirical performance under varying conditions. In order to further improve the efficiency of arc consistency enforcing, more properties (both general like bidirectionality and special like monotonicity) of constraints and heuristics are desirable.

[Wallace, 1993] gives detailed experiments comparing the efficiency of AC-3 and AC-4. Our work complements this in the sense that with the new implementation, AC-3 now has optimal worst case time complexity.

6 Conclusion

This paper presents a natural implementation of AC-3 whose complexity is better than the traditional understanding. AC-3 was not previously known to have worst case optimal time complexity even though it is known to be efficient. Our new implementation brings AC-3 to $\mathcal{O}(ed^2)$ on par with the other optimal worst case time complexity algorithms. Techniques in the new implementation can also be used with path consistency algorithms.

While worst case time complexity gives us the upper bound on the time complexity, in practice, the running time and number of constraint checks for various CSP instances are the prime consideration. Our preliminary experiments show that the new implementation significantly reduces the number of constraint checks and the running time of the traditional one on hard arc consistency problems. Furthermore, the running time of AC-3.1 is competitive with the known best algorithms based on the benchmarks from the experiment results in [Bessiere *et al.*, 1999]. Further experiments are planed to have a better comparison with typical algorithms. We believe that based on the CELAR instances, the new approach to AC-3 leads to a more robust AC algorithm for real world problems than other algorithms.

We also show how the new AC-3 leads to a new algorithm for path consistency. We conjecture from the results of [Chmeiss and Jegou, 1996] that this algorithm can give a practical implementation for path consistency.

For future work, we want to examine the new AC-3 in maintaining arc consistency during search.

7 Acknowledgment

We are grateful to Christian Bessiere for providing benchmarks and discussion. We acknowledge the generosity of the French Centre d'Electronique de l'Armement for providing the CELAR benchmarks.

References

[Bessiere, 1994] C. Bessiere 1994. Arc-consistency and arc-consistency again, *Art. Int.*65 (1994) 179–190.

[Bessiere *et al.*, 1999] C. Bessiere, E. C. Freuder and J. Regin 1999. Using constraint metaknowledge to reduce arc consistency computation, *Art. Int.*107 (1999) 125–148.

[Bessiere and Regin, 1996] C. Bessiere and J. Regin 1996. MAC and combined heuristics: two reasons to forsake FC(and CBJ?) on hard problems, *Proc. of Principles and Practice of Constraint Programming*, Cambridge, MA. pp. 61–75.

[Cabon *et al.*, 1999] B. Cabon, S. de Givry, L. Lobjois, T. Schiex and J.P. Warners 1999. Radio link frequency assignment, *Constraints* 4(1) (1999) 79–89.

[Chmeiss and Jegou, 1996] A. Chmeiss and P. Jegou 1996. Path-Consistency: When Space Misses Time, *Proc. of AAAI-96*, USA: AAAI press.

[Frost *et al.*, 1996] D. Frost, C. Bessiere, R. Dechter and J. C. Regin 1996. Random uniform CSP generators, `http://www.lirmm.fr/~bessiere/generator.html`.

[Gaschnig, 1978] J. Gaschnig 1978. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisfying assignment problems, *Proc. of CCSCSI-78*, 1978.

[Gent *et al.*, 1997] J. P. Gent, E. Maclntyre, P. Prosser, P. Shar and T. Walsh 1997. The constrainedness of arc consistency, *Proc. of Principles and Practice of Constraint Programming 1997* Cambridge, MA, 1996, pp. 327–340.

[Van Hentenryck *et al.*, 1992] P. van Hentenryck, Y. Deville, and C. M. Teng 1992. A Generic Arc-Consistency Algorithm and its Specializations, *Art. Int.*58 (1992) 291–321.

[Mackworth, 1977] A. K. Mackworth 1977. Consistency in Networks of Relations,, *Art. Int.*8(1) (1977) 118–126.

[Mackworth and Freuder, 1985] A. K. Mackworth and E. C. Freuder 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems,, *Art. Int.*25 (1985) 65–74.

[Mohr and Henderson, 1986] R. Mohr and T. C. Henderson 1986. Arc and Path Consistency Revisited, *Art. Int.*28 (1986) 225–233.

[Montanari, 1974] U. Montanari 1974. Networks of Constraints: Fundamental Properties and Applications, *Information Science* 7(2) (1974) 95–132.

[Singh, 1996] M. Singh 1996. Path consistency revisited, *Int. Journal on Art. Intelligence Tools* 5(1&2) (1996) 127–141.

[Wallace, 1993] Richard J. Wallace 1993. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs , *Proc. of IJCAI-93* Chambery, France, 1993.

[Wallace, 1992] R. J. Wallace and E. Freud 1992. Ordering heuristics for arc consistency algorithms , *Proc, of Canadian Conference on AI* Vancouver, BC, 1992, pp. 163-169.

[Zhang and Yap, 2000] Y. Zhang, R. H. C. Yap 2000. Arc consistency on n-ary monotonic and linear constraints, *Proc. of Principles and Practice of Constraint Programming* Singapore, 2000, pp. 470–483 .

Temporal Constraint Reasoning With Preferences

Lina Khatib[1,2] **Paul Morris**[2] **Robert Morris**[2]
1. Kestrel Technology
2. Computational Sciences Division
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035

Francesca Rossi
Dipartimento di Matematica Pura ed Applicata
Universita' di Padova
Via Belzoni 7, 35131 Padova, Italy

Abstract

A number of reasoning problems involving the manipulation of temporal information can be viewed as implicitly inducing an ordering of decisions involving time (associated with durations or orderings of events) on the basis of preferences. For example, a pair of events might be constrained to occur in a certain order, and, in addition, it might be preferable that the delay between them be as large, or as small, as possible. This paper explores problems in which a set of temporal constraints is specified, each with preference criteria for making local decisions about the events involved in the constraint. A reasoner must infer a complete solution to the problem such that, to the extent possible, these local preferences are met in the best way. Constraint-based temporal reasoning is generalized to allow for reasoning about temporal preferences, and the complexity of the resulting formalism is examined. While in general such problems are NP-complete, some restrictions on the shape of the preference functions, and on the structure of the set of preference values, can be enforced to achieve tractability. In these cases, a generalization of a single-source shortest path algorithm can be used to compute a globally preferred solution in polynomial time.

1 Introduction and Motivation

Some real world temporal reasoning problems can naturally be viewed as involving preferences associated with decisions such as how long a single activity should last, when it should occur, or how it should be ordered with respect to other activities. For example, an antenna on an earth orbiting satellite such as Landsat 7 must be slewed so that it is pointing at a ground station in order for recorded science data to be downlinked to earth. Assume that as part of the daily Landsat 7 scheduling activity a window $W = [s, e]$ is identified within which a slewing activity to one of the ground stations for one of the antennae can begin, and thus there are choices for assigning the start time for this activity. Antenna slewing on Landsat 7 has been shown to cause a vibration to the satellite, which in turn affects the quality of the observation taken by the imaging instrument if the instrument is in use during slewing. Consequently, it is preferable for the slewing activity not to overlap any scanning activity, although because the detrimental effect on image quality occurs only intermittently, this disjointness is best not expressed as a hard constraint. Rather, the constraint is better expressed as follows: if there are any start times t within W such that no scanning activity occurs during the slewing activity starting at t, then t is to be preferred. Of course, the cascading effects of the decision to assign t on the sequencing of other satellite activities must be taken into account as well. For example, the selection of t, rather than some earlier start time within W, might result in a smaller overall contact period between the ground station and satellite, which in turn might limit the amount of data that can be downlinked during this period. This may conflict with the preference for maintaining maximal contact times with ground stations.

Reasoning simultaneously with hard temporal constraints and preferences, as illustrated in the example just given, is the subject of this paper. The overall objective is to develop a system that will generate solutions to temporal reasoning problems that are *globally preferred* in the sense that the solutions simultaneously meet, to the best extent possible, all the local preference criteria expressed in the problem.

In what follows a formalism is described for reasoning about temporal preferences. This formalism is based on a generalization of the Temporal Constraint Satisfaction Problem (TCSP) framework [Dechter *et al*, 1991], with the addition of a mechanism for specifying preferences, based on the semiring-based soft constraint formalism [Bistarelli *et. al.*, 1997] . The result is a framework for defining problems involving *soft temporal constraints*. The resulting formulation, called Temporal Constraint Satisfaction Problems with Preferences (TCSPPs) is introduced in Section 2. A sub-class of TCSPPs in which each constraint involves only a single interval, called Simple Temporal Problems with Preferences (STPPs), is also defined. In Section 3, we demonstrate the hardness of solving general TCSPPs and STPPs, and pinpoint one source of the hardness to preference functions whose "better" values may form a non-convex set. Restricting the class of admissible preference functions to those with convex intervals of "better" values is consequently shown to result in a tractable framework for solving STPPs. In section 4, an algorithm is introduced, based on a simple generalization of

the single source shortest path algorithm, for finding globally best solutions to STPPs with restricted preference functions. In section 5, the work presented here is compared to other approaches and results.

2 Temporal Constraint Problems with Preferences

The proposed framework is based on a simple merger of two existing formalisms: Temporal Constraint Satisfaction Problems (TCSPs) [Dechter *et. al.*, 1991], and soft constraints based on semirings [Bistarelli *et. al.*, 1997] [1]. The result of the merger is a class of problems called Temporal Constraint Satisfaction problems with preferences (TCSPPs). In a TCSPP, a *soft temporal constraint* is represented by a pair consisting of a set of disjoint intervals and a preference function: $\langle I = \{[a_1, b_1], \ldots, [a_n, b_n]\}, f\rangle$, where $f : I \rightarrow A$, and A is a set of preference values.

Examples of preference functions involving time are:

- **min-delay**: any function in which smaller distances are preferred, that is, the delay of the second event w.r.t. the first one is minimized.
- **max-delay**: assigning higher preference values to larger distances;
- **close to k**: assign higher values to distances which are closer to k; in this way, we specify that the distance between the two events must be as close as possible to k.

As with classical TCSPs, the interval component of a soft temporal constraint depicts restrictions either on the start times of events (in which case they are unary), or on the distance between pairs of distinct events (in which case they are binary). For example, a unary constraint over a variable X representing an event, restricts the domain of X, representing its possible times of occurrence; then the interval constraint is shorthand for $(a_1 \leq X \leq b_1) \vee \ldots \vee (a_n \leq X \leq b_n)$. A binary constraint over X and Y, restricts the values of the distance $Y - X$, in which case the constraint can be expressed as $(a_1 \leq Y - X \leq b_1) \vee \ldots \vee (a_n \leq Y - X \leq b_n)$. A uniform, binary representation of all the constraints results from introducing a variable X_0 for the *beginning of time*, and recasting unary constraints as binary constraints involving the distance $X - X_0$.

An interesting special case occurs when each constraint of a TCSPP contains a single interval. We call such problems *Simple Temporal Problems with Preferences* (STPPs), due to the fact that they generalize STPs [Dechter *et. al.*, 1991]. This case is interesting because STPs are polynomially solvable, while general TCSPs are NP-complete, and the effect of adding preferences to STPs is not immediately obvious. The next section discusses these issues in more depth.

A *solution* to a TCSPP is a complete assignment to all the variables that satisfies the distance constraints. Each solution has a *global preference value*, obtained by combining the local preference values found in the constraints. To formalize the process of combining local preferences into a global preference, and comparing solutions, we impose a semiring structure onto the TCSPP framework.

A *semiring* is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that

- A is a set and $\mathbf{0}, \mathbf{1} \in A$;
- $+$, the additive operation, is commutative, associative and $\mathbf{0}$ is its unit element;
- $\times$, the multiplicative operation, is associative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element.

A *c-semiring* is a semiring in which $+$ is idempotent (i.e., $a + a = a, a \in A$), $\mathbf{1}$ is its absorbing element, and $\times$ is commutative.

C-semirings allow for a partial order relation $\leq_S$ over A to be defined as $a \leq_S b$ iff $a + b = b$. Informally, $\leq_S$ gives us a way to compare tuples of values and constraints, and $a \leq_S b$ can be read *b is better than a*. Moreover: $+$ and $\times$ are monotone on $\leq_S$; $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum; $\langle A, \leq_S\rangle$ is a complete lattice and, for all $a, b \in A$, $a + b = lub(a, b)$ (where *lub*=least upper bound). If $\times$ is idempotent, then $\langle A, \leq_S\rangle$ is a complete distributive lattice and $\times$ is its greatest lower bound (*glb*). In our main results, we will assume $\times$ is idempotent and also restrict $\leq_S$ to be a total order on the elements of A. In this case $a + b = \max(a, b)$ and $a \times b = \min(a, b)$.

Given a choice of semiring with a set of values A, each preference function f associated with a soft constraint $\langle I, f\rangle$ takes an element from I and returns an element of A. The semiring operations allow for complete solutions to be evaluated in terms of the preference values assigned locally. More precisely, given a solution t in a TCSPP with associated semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, let $T_{ij} = \langle I_{i,j}, f_{i,j}\rangle$ be a soft constraint over variables X_i, X_j and (v_i, v_j) be the projection of t over the values assigned to variables X_i and X_j (abbreviated as $(v_i, v_j) = t_{\downarrow X_i, X_j}$). Then, the corresponding preference value given by f_{ij} is $f_{ij}(v_j - v_i)$, where $v_j - v_i \in I_{i,j}$. Finally, where $F = \{x_1, \ldots, x_k\}$ is a set, and $\times$ is the multiplicative operator on the semiring, let $\times F$ abbreviate $x_1 \times \ldots \times x_k$. Then the global preference value of t, $val(t)$, is defined to be $val(t) = \times\{f_{ij}(v_j - v_i) \mid (v_i, v_j) = t_{\downarrow X_i, X_j}\}$.

The optimal solutions of a TCSPP are those solutions which have the best preference value, where "best" is determined by the ordering of the values in the semiring. For example, consider the semiring $S_{fuzzy} = \langle[0, 1], max, min, 0, 1\rangle$, used for fuzzy constraint solving [Schiex, 1995]. The preference value of a solution will be the minimum of all the preference values associated with the distances selected by this solution in all constraints, and the best solutions will be those with the maximal value. Another example is the semiring $S_{csp} = \langle\{false, true\}, \vee, \wedge, false, true\rangle$, which is related to solving classical constraint problems [Mackworth, 1992]. Here there are only two preference values: *true* and *false*, the preference value of a complete solution will be determined by the logical *and* of all the local preferences, and the best solutions will be those with preference value *true* (since *true* is better

[1] Semiring-based soft constraints is one of a number of formalisms for soft constraints, but it has been shown to generalize many of the others, e.g., [Freuder and Wallace, 1992] and [Schiex *et. al.*, 1995].

than *false* in the order induced by logical or). This semiring thus recasts the classical TCSP framework into a TCSPP.

Given a constraint network, it is often useful to find the corresponding minimal network in which the constraints are as explicit as possible. This task is normally performed by enforcing various levels of local consistency. For TCSPPs, in particular, we can define a notion of *path consistency*. Given two soft constraints, $\langle I_1, f_1\rangle$ and $\langle I_2, f_2\rangle$, and a semiring S, we define:

- the *intersection* of two soft constraints $T_1 = \langle I_1, f_1\rangle$ and $T_2 = \langle I_2, f_2\rangle$, written $T_1 \oplus_S T_2$, as the soft constraint $\langle I_1 \oplus I_2, f\rangle$, where
  - $I_1 \oplus I_2$ returns the pairwise intersection of intervals in I_1 and I_2, and
  - $f(a) = f_1(a) \times f_2(a)$ for all $a \in I_1 \oplus I_2$;
- the *composition* of two soft constraints $T_1 = \langle I_1, f_1\rangle$ and $T_2 = \langle I_2, f_2\rangle$, written $T_1 \otimes_S T_2$, is the soft constraint $T = \langle I_1 \otimes I_2, f\rangle$, where
  - $r \in I_1 \otimes I_2$ if and only if there exists a value $t_1 \in I_1$ and $t_2 \in I_2$ such that $r = t_1 + t_2$, and
  - $f(a) = \sum\{f_1(a_1) \times f_2(a_2) \mid a = a_1 + a_2, a_1 \in I_1, a_2 \in I_2\}$, where $\sum$ is the generalization of $+$ over sets.

A *path-induced* constraint on variables X_i and X_j is $R_{ij}^{path} = \oplus_S \forall k (T_{ik} \otimes T_{kj})$, i.e., the result of performing $\oplus_S$ on each way of composing paths of size two between i and j. A constraint T_{ij} is *path-consistent* if and only if $T_{ij} \subseteq R_{ij}^{path}$, i.e., T_{ij} is at least as strict as R_{ij}^{path}. A TCSPP is path-consistent if and only if all its constraints are path-consistent.

If the multiplicative operation of the semiring is idempotent, then it is easy to prove that applying the operation $T_{ij} := T_{ij} \oplus_S (T_{ik} \otimes_S T_{kj})$ to any constraint T_{ij} of a TCSPP returns an equivalent TCSPP. Moreover, under the same condition, applying this operation to a set of constraints returns a final TCSPP which is always the same independently of the order of application[2]. Thus any TCSPP can be transformed into an equivalent path-consistent TCSPP by applying the operation $T_{ij} := T_{ij} \oplus (T_{ik} \otimes T_{kj})$ to all constraints T_{ij} until no change occurs in any constraint. This algorithm, which we call Path, is proven to be polynomial for TCSPs (that is, TCSPPs with the semiring S_{csp}): its complexity is $O(n^3 R^3)$, where n is the number of variables and R is the range of the constraints [Dechter *et. al.*, 1991].

General TCSPPs over the semiring S_{csp} are NP-complete; thus applying Path is insufficient to solve them. On the other hand, with STPPs over the same semiring that coincide with STPs, applying Path is sufficient to solve them. In the remaining sections, we prove complexity results for both general TCSPPs and STPPs, and also of some subclasses of problems identified by specific semirings, or preference functions with a certain shape.

[2]These properties are trivial extensions of corresponding properties for classical CSPs, proved in [Bistarellli, *et. al.*, 1997.]

3 Solving TCSPPs and STPPs is NP-Complete

As noted above, solving TCSPs is NP-Complete. Since the addition of preference functions can only make the problem of finding the optimal solutions more complex, it is obvious that TCSPPs are at least NP-Complete as well.

We turn our attention to the complexity of general STPPs. We recall that STPs are polynomially solvable, thus one might speculate that the same is true for STPPs. However, it is possible to show that in general, STPPs fall into the class of NP-Complete problems.

Theorem 1 (complexity of STPPs) *General STPPs are NP-complete problems.*

Proof:

First, we prove that STPPs belong to NP. Given an instance of the feasibility version of the problem, in which we wish to determine whether there is a solution to the STTP with global preference value $\geq k$, for some k, we use as a certificate the set of times assigned to each event. The verification algorithm "chops" the set of preference values of each local preference function at k. The result of a chop, for each constraint, is a set of intervals of temporal values whose preference values are greater than k. The remainder of the verification process reduces to the problem of verifying General Temporal CSPs (TCSP), which is done by non-deterministically choosing an interval on each edge of the TCSP, and solving the resulting STP, which can be done in polynomial time. Therefore, STTPs belong to NP.

To prove hardness we reduce an arbitrary TCSP to an STPP. Thus, consider any TCSP, and take any of its constraints, say $I = \{[a_1, b_1], \ldots, [a_n, b_n]\}$. We will now obtain a corresponding soft temporal constraint containing just one interval (thus belonging to an STPP). The semiring that we will use for the resulting STPP is the classical one: $S_{csp} = \langle\{false, true\}, \vee, \wedge, false, true\rangle$. Thus the only two allowed preference values are false and true (or 0 and 1). Assuming that the intervals in I are ordered such that $a_i \leq a_{i+1}$ for $i \in \{1, \ldots, n-1\}$, the interval of the soft constraint is just $[a_1, b_n]$. The preference function will give value 1 to values in I and 0 to the others. Thus we have obtained an STPP whose set of solutions with value 1 (which are the optimal solutions, since $0 \leq_S 1$ in the chosen semiring) coincides with the set of solutions of the given TCSP. Since finding the set of solutions of a TCSP is NP-hard, it follows that the problem of finding the set of optimal solutions to an STPP is NP-hard. □

4 Linear and Semi-Convex Preference Functions

The hardness result for STPPs derives either from the nature of the semiring or the shape of the preference functions. In this section, we identify classes of preference functions which define tractable subclasses of STPPs.

When the preference functions of an STPP are linear, and the semiring chosen is such that its two operations maintain such linearity when applied to the initial preference functions, the given STPP can be written as a linear programming problem, solving which is tractable [Cormen *et. al.*, 1990]. Thus,

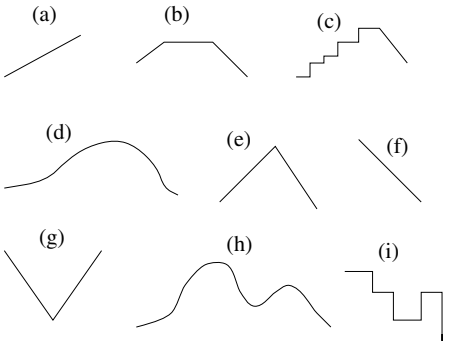

Figure 1: Examples of semi-convex functions (a)-(f) and non-semi-convex functions (g)-(i)

consider any given TCSPP. For any pair of variables X and Y, take each interval for the constraint over X and Y, say $[a, b]$, with associated linear preference function f. The information given by each of such intervals can be represented by the following inequalities and equation: $X - Y \leq b$, $Y - X \leq -a$, and $f_{X,Y} = c_1(X - Y) + c_2$. Then if we choose the fuzzy semiring $\langle [0, 1], max, min, 0, 1\rangle$, the global preference value V will satisfy the inequality $V \leq f_{X,Y}$ for each preference function $f_{X,Y}$ defined in the problem, and the objective is $max(V)$. If instead we choose the semiring $\langle \mathcal{R}, min, +, \infty, 0\rangle$, where the objective is to minimize the sum of the preference levels, we have $V = f_1 + \ldots + f_n$ and the objective is $min(V)$ [3]. In both cases the resulting set of formulas constitutes a linear programming problem.

Linear preference functions are expressive enough for many cases, but there are also several situations in which we need preference functions which are not linear. A typical example arises when we want to state that the distance between two variables must be as close as possible to a single value. Unless this value is one of the extremes of the interval, the preference function is convex, but not linear. Another case is one in which preferred values are as close as possible to a single distance value, but in which there are some subintervals where all values have the same preference. In this case, the preference criteria define a *step function*, which is not convex.

A class of functions which includes linear, convex, and also some step functions will be called *semi-convex functions*. Semi-Convex functions have the property that if one draws a horizontal line anywhere in the Cartesian plane defined by the function, the set of X such that $f(X)$ is not below the line forms an interval. Figure 1 shows examples of semi-convex and non-semi-convex functions.

More formally, a *semi-convex function* is one such that, for all Y, the set $\{X$ such that $f(X) \geq Y\}$ forms an interval. It is easy to see that semi-convex functions include linear ones, as well as convex and some step functions. For example, the *close to k* criteria cannot be coded into a linear preference function, but it can be specified by a semi-convex preference function, which could be $f(x) = x$ for $x \leq k$ and $f(x) = 2k - x$ for $x > k$.

[3] In this context, the "+" is to be interpreted as the arithmetic operation, not the additive operation of the semiring.

Semi-Convex functions are closed under the operations of intersection and composition defined in Section 2, when certain semirings are chosen. For example, this happens with the fuzzy semiring, where the intersection performs the *min*, and composition performs the *max* operation. The closure proofs follow.

Theorem 2 (closure under intersection) *The property of functions being semi-convex is preserved under intersection. That is, given a totally-ordered semiring with an idempotent multiplicative operation $\times$ and binary additive operation $+$ (or $\sum$ over an arbitrary set of elements), let f_1 and f_2 be semi-convex functions which return values over the semiring. Let f be defined as $f(a) = f_1(a) \times f_2(a)$, where $\times$ is the multiplicative operation of the semiring. Then f is a semi-convex function as well.*

Proof: From the definition of semi-convex functions, it suffices to prove that, for any given y, the set $S = \{x : f(x) \geq y\}$ identifies an interval. If S is empty, then it identifies the empty interval. In the following we assume S to be not empty.

$$\begin{aligned}\{x : f(x) \geq y\} &= \{x : f_1(x) \times f_2(x) \geq y\} \\ &= \{x : min(f_1(x), f_2(x)) \geq y\}\end{aligned}$$

($\times$ is a lower bound operator since it is assumed to be idempotent)

$$\begin{aligned}&= \{x : f_1(x) \geq y \wedge f_2(x) \geq y\} \\ &= \{x : x \in [a_1, b_1] \wedge x \in [a_2, b_2]\}\end{aligned}$$

(since each of f_1 and f_2 is semi-convex)

$$= [max(a_1, a_2), min(b_1, b_2)]$$

□

Theorem 3 (closure under composition) *The property of functions being semi-convex is preserved under composition. That is, given a totally-ordered semiring with an idempotent multiplicative operation $\times$ and binary additive operation $+$ (or $\sum$ over an arbitrary set of elements), let f_1 and f_2 be semi-convex functions which return values over the semiring. Define f as $f(a) = \sum_{b+c=a}(f_1(b) \times f_2(c))$. Then f is a semi-convex function as well.*

Proof: Again, from the definition of semi-convex functions, it suffices to prove that, for any given y, the set $S = \{x : f(x) \geq y\}$ identifies an interval. If S is empty, then it identifies the empty interval. In the following we assume S to be not empty.

$$\begin{aligned}&\{x : f(x) \geq y\} \\ &\quad = \{x : \textstyle\sum_{u+v=x}(f_1(u) \times f_2(v)) \geq y\} \\ &\quad = \{x : \max_{u+v=x}(f_1(u) \times f_2(v)) \geq y\}\end{aligned}$$

(since $+$ is an upper bound operator)

$$\begin{aligned}&\quad = \{x : f_1(u) \times f_2(v) \geq y \text{ for some } u \text{ and } v \\ &\qquad\qquad \text{such that } x = u + v\} \\ &\quad = \{x : \min(f_1(u), f_2(v)) \geq y \text{ for some } u \text{ and } v \\ &\qquad\qquad \text{such that } x = u + v\}\end{aligned}$$

($\times$ is a lower bound operator since it is assumed to be idempotent)

$$= \{x : f_1(u) \geq y \wedge f_2(v) \geq y,$$

for some $u + v = x\}$
$= \{x : u \in [a_1, b_1] \wedge v \in [a_2, b_2],$
for some $u + v = x$ and some $a_1, b_1, a_2, b_2\}$
(since each of f_1 and f_2 is semi-convex)
$= \{x : x \in [a_1 + a_2, b_1 + b_2]\}$
$= [a_1 + a_2, b_1 + b_2]$

□

That closure of the set of semi-convex functions requires a total order and idempotence of the $\times$ operator is demonstrated by the following example. In what follows we assume monotonicity of the $\times$ operator. Let **a** and **b** be preference values with $\mathbf{a} \not\leq \mathbf{b}$, $\mathbf{b} \not\leq \mathbf{a}$, $\mathbf{a} \times \mathbf{b} < \mathbf{a}$, and $\mathbf{a} \times \mathbf{b} < \mathbf{b}$. Suppose x_1 and x_2 are real numbers with $x_1 < x_2$. Define $g(x) = \mathbf{1}$ for $x < x_1$ and $g(x) = \mathbf{a}$ otherwise. Also define $h(x) = \mathbf{b}$ for $x < x_2$ and $h(x) = \mathbf{1}$ otherwise. Clearly, g and h are semi-convex functions. Define $f = g \times h$. Note that $f(x) = \mathbf{b}$ for $x < x_1$, $f(x) = \mathbf{a} \times \mathbf{b}$ for $x_1 \leq x < x_2$ and $f(x) = \mathbf{a}$ for $x \geq x_2$. Since $\{x | f(x) \not\leq a\}$ includes all values except where $x_1 \leq x < x_2$, f is not semi-convex.

Now consider the situation where the partial order is not total. Then there are distinct incomparable values **a** and **b** that satisfy the condition of the example. We conclude the order must be total. Next consider the case in which idempotence is not satisfied. Then there is a preference value **c** such that $\mathbf{c} \times \mathbf{c} \neq \mathbf{c}$. It follows that $\mathbf{c} \times \mathbf{c} < \mathbf{c}$. In this case, setting $\mathbf{a} = \mathbf{b} = \mathbf{c}$ satisfies the condition of the example. We conclude that idempotence is also required.

The results in this section imply that applying the Path algorithm to an STPP with only semi-convex preference functions, and whose underlying semiring contains a multiplicative operation that is idempotent, and whose values are totally ordered, will result in a network whose induced soft constraints also contain semi-convex preference functions. These results will be applied in the next section.

5 Solving STPPs with Semi-Convex Functions is Tractable

We will now prove that STPPs with semi-convex preference functions and an underlying semiring with an idempotent multiplicative operation can be solved tractably.

First, we describe a way of transforming an arbitrary STPP with semi-convex preference functions into a STP. Given an STPP and an underlying semiring with A the set of preference values, let $y \in A$ and $\langle I, f \rangle$ be a soft constraint defined on variables X_i, X_j in the STPP, where f is semi-convex. Consider the interval defined by $\{x : x \in I \wedge f(x) \geq y\}$ (because f is semi-convex, this set defines an interval for any choice of y). Let this interval define a constraint on the same pair X_i, X_j. Performing this transformation on each soft constraint in the original STPP results in an STP, which we refer to as STP_y. (Notice that not every choice of y will yield an STP that is solvable.) Let *opt* be the highest preference value (in the ordering induced by the semiring) such that STP_{opt} has a solution. We will now prove that the solutions of STP_{opt} are the optimal solutions of the given STPP.

Theorem 4 *Consider any STPP with semi-convex preference functions over a totally-ordered semiring with $\times$ idempotent. Take opt as the highest y such that STP_y has a solution. Then the solutions of STP_{opt} are the optimal solutions of the STPP.*

Proof: First we prove that every solution of STP_{opt} is an optimal solution of STPP. Take any solution of STP_{opt}, say t. This instantiation t, in the original STPP, has value $val(t) = f_1(t_1) \times ... \times f_n(t_n)$, where t_i is the distance $v_j - v_i$ for an assignment to the variables X_i, X_j, $(v_i, v_j) = t \downarrow_{X_i, X_j}$, and f_i is the preference function associated with the soft constraint $\langle I_i, f_i \rangle$, with $v_j - v_i \in I_i$.

Now assume for the purpose of contradiction that t is not optimal in STPP. That is, there is another instantiation t' such that $val(t') > val(t)$. Since $val(t') = f_1(t'_1) \times ... \times f_n(t'_n)$, by monotonicity of the $\times$, we can have $val(t') > val(t)$ only if each of the $f_i(t'_i)$ is greater than the corresponding $f_i(t_i)$. But this means that we can take the smallest such value $f_i(t'_i)$, call it w', and construct $STP_{w'}$. It is easy to see that $STP_{w'}$ has at least one solution, t', therefore *opt* is not the highest value of y, contradicting our assumption.

Next we prove that every optimal solution of the STPP is a solution of STP_{opt}. Take any t optimal for STPP, and assume it is not a solution of STP_{opt}. This means that, for some constraint, $f(t_i) < opt$. Therefore, if we compute $val(t)$ in STPP, we have that $val(t) < opt$. Then take any solution t' of STP_{opt} (there are some, by construction of STP_{opt}). If we compute $val(t')$ in STPP, since $\times$ = glb (we assume $\times$ idempotent), we have that $val(t') \geq opt$, thus t was not optimal as initially assumed. □

This result implies that finding an optimal solution of the given STPP with semi-convex preference functions reduces to a two-step search process consisting of iteratively choosing a w, then solving STP_w, until STP_{opt} is found. Under certain conditions, both phases can be performed in polynomial time, and hence the entire process can be tractable.

The first phase can be conducted naively by trying every possible "chop" point y and checking whether STP_y has a solution. A binary search is also possible. Under certain conditions, it is possible to see that the number of chop points is also polynomial, namely:

- if the semiring has a finite number of elements, which is at most exponential in the number n of variables of the given STPP, then a polynomial number of checks is enough using binary search.
- if the semiring has a countably infinite number of elements, and the preference functions never go to infinity, then let l be the highest preference level given by the functions. If the number of values not above l is at most exponential in n, then again we can find *opt* in a polynomial number of steps.

The second phase, solving the induced STP_y, can be performed by transforming the graph associated with this STP into a distance graph, then solving two single-source shortest path problems on the distance graph [Dechter *et. al.*, 1991]. If the problem has a solution, then for each event it is possible to arbitrarily pick a time within its time bounds, and find corresponding times for the other events such that the set of times for all the events satisfy the interval constraints. The

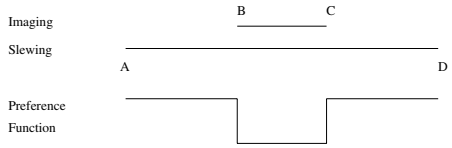

Figure 2: Non-semi-convex Preference Function for the Landsat problem

complexity of this phase is $O(en)$ (using the Bellman-Ford algorithm [Cormen *et. al.*, 1990]).

The main result of this discussion is that, while general TCSPPs are NP-Complete, there are sub-classes of TCSPP problems which are polynomially solvable. Important sources of tractability include the shape of the temporal preference functions, and the choice of the underlying semiring for constructing and comparing preference values.

Despite this encouraging theoretical result, the extent to which real world preferences conform to the conditions necessary to utilize the result is not clear. To illustrate this, consider again the motivating example at the outset. As illustrated in Figure 2, suppose an imaging event is constrained to occur during $[B, C]$, and that the interval $[A, D]$ is the interval during which a slewing event can start to occur. Assuming the soft constraint that prefers no overlap between the two occurrences, the preference values for the slewing can be visualized as the function pictured below the interval, a function that is not semi-convex. A semi-convex preference function would result by squeezing one or the other of the endpoints of the possible slewing times far enough that the interval would no longer contain the imaging time. For example, removing the initial segment $[A, B]$ from the interval of slewing times would result in a semi-convex preference function. Dealing with the general case in which preference functions are not semi-convex is a topic of future work.

6 Related work

The merging of temporal CSPs with soft constraints was first proposed in [Morris and Khatib, 2000], where it was used within a framework for reasoning about recurring events. The framework proposed in [Rabideau *et. al.*, 2000] contains a representation of local preferences that is similar to the one proposed here, but uses local search, rather than constraint propagation, as the primary mechanism for finding good complete solutions, and no guarantee of optimality can be demonstrated.

Finally, the property that characterizes semi-convex preference functions, viz., the convexity of the interval above any horizontal line drawn in the Cartesian plane around the function, is reminiscent of the notion of row-convexity, used in characterizing constraint networks whose global consistency, and hence tractability in solving, can be determined by applying local (path) consistency [Van Beek and Dechter, 1995]. There are a number of ways to view this connection. One way is to note that the row convex condition for the 0-1 matrix representation of binary constraints prohibits a row in which a sequence of ones is interrupted by one or more zeros. Replacing the ones in the matrix by the preference value for that pair of domain elements, one can generalize the definition of row convexity to prohibit rows in which the preference values decrease then increase. This is the intuitive idea underlying the behavior of semi-convex preference functions.

7 Summary

We have defined a formalism for characterizing problems involving temporal constraints over the distances and duration of certain events, as well as preferences over such distances. This formalism merges two existing frameworks, temporal CSPs and soft constraints, and inherits from them their generality, and also allows for a rigorous examination of computational properties that result from the merger.

Acknowledgments

Thanks to David McAllister, Rina Dechter and Alessandro Sperduti for helpful comments in earlier stages of the development of this work. This work was partially funded by the Research Institute for Advanced Computer Science, NASA Ames Research Center, and by the EC TMR Network GETGRATS (General Theory of Graph Transformation Systems).

References

[Bistarelli *et al.*, 1997] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, March 1997.

[Cormen *et al.*, 1990] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT press, Cambridge, MA, 1990.

[Dechter *et al.*, 1991] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49, 1991:61–95, 1991.

[Freuder and Wallace , 1992] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58, 1992.

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

[Schiex *et al.*, 1995] T. Schiex, H. Fargier, and G. Verfaille. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proceedings of IJCAI95*, pages 631–637. Morgan Kaufmann, 1995.

[Morris and Khatib , 2000] R. Morris, and L. Khatib. Reasoning about Recurring Events: Satisfaction and Optimization. In *Computational Intelligence*, 16(2), 2000.

[Rabideau *et al.*, 2000] G. Rabideau, B. Engelhardt and S. Chien. Using Generic Preferences to Incrementally Improve Plan Quality. In *Proceedings of the 2nd NASA International Workshop on Planning and Scheduling for Space*, 11–16, March, 2000.

[Van Beek and Dechter , 1995] P. Van Beek, and R. Dechter. On the Minimality and Global Consistency of Row-Convex Constraint Networks. In *Journal of the ACM*, 42, 543–561, 1995.

A Hybrid Approach for the 0–1 Multidimensional Knapsack problem

Michel Vasquez 1) and Jin-Kao Hao 2)
1) EMA–EERIE, Parc Scientifique G. Besse, F–30035 Nimes cedex 1, vasquez@site-eerie.ema.fr
2) Université d'Angers, 2 bd Lavoisier, F–49045 Angers cedex 1, hao@info.univ-angers.fr

Abstract

We present a hybrid approach for the 0–1 multidimensional knapsack problem. The proposed approach combines linear programming and Tabu Search. The resulting algorithm improves significantly on the best known results of a set of more than 150 benchmark instances.

1 Introduction

The NP-hard 0–1 multidimensional knapsack problem (MKP01) consists in selecting a subset of n given objects (or items) in such a way that the total profit of the selected objects is maximized while a set of knapsack constraints are satisfied. More formally, the MKP01 can be stated as follows.

$$\text{MKP01} \left\{ \begin{array}{l} \text{maximize } c.x \text{ subject to} \\ A.x \leq b \text{ and } x \in \{0,1\}^n \end{array} \right.$$

where $c \in \mathbb{N}^{*n}$, $A \in \mathbb{N}^{m \times n}$ and $b \in \mathbb{N}^{m}$. The binary components x_j of x are decision variables: $x_j = 1$ if the object j is selected, 0 otherwise. c_j is the profit associated to j. Each of the m constraints $A_i.x \leq b_i$ is called a knapsack constraint.

The special case of the MKP01 with $m = 1$ is the classical knapsack problem (KP01), whose usual statement is the following. Given a knapsack of capacity b and n objects, each being associated a profit (gain) and a volume occupation, one wants to select k ($k \leq n$ and k not fixed) objects such that the total profit is maximized and the capacity of the knapsack is not exceeded. It is well known that the KP01 is not strongly NP-hard because there are polynomial approximation algorithms to solve it. This is not the case for the general MKP01.

The MKP01 can formulate many practical problems such as capital budgeting where project j has profit c_j and consume (a_{ij}) units of resource i. The goal is to determine a subset of the n projects such that the total profit is maximized and all resource constraints are satisfied. Other important applications include cargo loading [Shih, 1979], cutting stock problems, and processor allocation in distributed systems [Gavish and Pirkul, 1982]. Notice that the MKP01 can be considered as a general 0–1 integer programming problem with non-negative coefficients.

Given the practical and theoretical importance of the MKP01, it is not surprising to find a large number of studies in the literature; we give a brief review of these studies in the next section.

2 State of the Art

Like for many NP-hard combinatorial optimization problems, both exact and heuristic algorithms have been developed for the MKP01. Existing exact algorithms are essentially based on the branch and bound method [Shih, 1979]. These algorithms are different one from another according to the way the upper bounds are obtained. For instance, in [Shih, 1979], Shih solves, exactly, each of the m single constrained, relaxed knapsack problems and select the minimum of the m objective function values as the upper bound. Better algorithms have been proposed by using tighter upper bounds, obtained with other MKP01 relaxation techniques such as lagrangean, surrogate and composite relaxations [Gavish and Pirkul, 1985]. Due to their exponential time complexity, exact algorithms are limited to small size instances ($n = 200$ and $m = 5$).

Heuristic algorithms are designed to produce near-optimal solutions for larger problem instances. The first heuristic approach for the MKP01 concerns for a large part greedy methods. These algorithms construct a solution by adding, according to a greedy criterion, one object each time into a current solution without constraint violation. The second heuristic approach is based on linear programming by solving various relaxations of the MKP01. A first possibility is to relaxe the integrality constraints and solve optimally the relaxed problem with simplex. Other possibilities include surrogate and composite relaxations [Osorio *et al.*, 2000].

More recently, several algorithms based on metaheuristics have been developed, including simulated annealing [Drexl, 1988], tabu search [Glover and Kochenberger, 1996; Hanafi and Fréville, 1998] and genetic algorithms [Chu and Beasley, 1998]. The metaheuristic approach has allowed to obtain very competitive results on large instances compared with other methods ($n = 500$ and $m = 30$).

Most of the above heuristics use the so-called *pseudo–utility* criterion for selecting the objects to be added into a solution. For the single constraint case (the KP01), this criterion corresponds to the *profit/resource* ratio. For the general MKP01, the *pseudo–utility* is usually defined as $c_j/(u.a_j)$

where u is a multiplier for the column vector x_j. Notice that it is impossible to obtain "optimal" multipliers. Thus the *pseudo–utility* criterion may mislead the search.

In this paper, we propose an alternative way to explore the search space. We use fractional optimal solutions given by linear programming to guide a neighborhood search (Tabu Search or TS) algorithm. The main idea is to exploit around a fractional optimal solution with additional constraints. We experiment this approach on a very large variety of MKP01 benchmark instances and compare our results with the best known ones. We show that this hybrid approach outperforms previously best algorithms for the set of tested instances.

The paper is organized as follows. Next section presents the general scope of our approach. Section 4 describes the algorithm used to determine *promising* sub-space of the whole search space from where we run a TS algorithm. This algorithm is presented in section 5. Finally we give computational results on a large set of MKP01 instances in section 6.

3 A Hybrid Approach for the MKP01

The basic idea of our approach is to search around the fractional optimum $\bar{x}$ of some relaxed MKP. Our hypothesis is that the search space around $\bar{x}$ should contain high quality solutions[1].

This general principle can be put into practice via a two phase approach. The first phase consists in solving exactly a relaxation MKP of the initial MKP01 to find a fractional optimal solution $\bar{x}$. The second phase consists in exploring carefully and efficiently some areas around this fractional optimal point.

Clearly, there are many possible ways to implement this general approach. For this work, we have done the following choices. The first phase is carried out by using simplex to solve a relaxed MKP. The second phase is ensured by a Tabu Search algorithm.

Let us notice first that relaxing the integrality constraints alone may not be sufficient since its optimal solution may be far away from an optimal binary solution. To be convinced, let us consider the following small example with five objects and one knapsack constraint:

$$sample \left\{ \begin{array}{lcl} c & = & (12 \quad 12 \quad 9 \quad 8 \quad 8) \\ A & = & (11 \quad 12 \quad 10 \quad 10 \quad 10) \quad b = 30 \end{array} \right.$$

This relaxed problem leads to the fractional optimal solution $\bar{x} = (1, 1, \frac{7}{10}, 0, 0)$ with an optimal cost value $\bar{z} = 30.3$ while the optimal binary solution is $x^{opt} = (0, 0, 1, 1, 1)$ with an optimal cost value $z^{opt} = 25$.

However, the above relaxation can be enforced to give more precise information regarding the discrete aspect of the problem. To do this, let us notice that all solutions of MKP01 verify the property: $\sum_{j=1}^{n} x_j = k \in \mathbb{N}$ with k an integer. Now if we add this constraint to the relaxed MKP, we obtain a series of problems:

$$\text{MKP}[k] \left\{ \begin{array}{l} \text{maximize } c.x \text{ s.t.} \\ A.x \leq b \text{ and } x \in [0\text{–}1]^n \text{ and} \\ \sigma(x) = k \in \mathbb{N} \end{array} \right.$$

where $\sigma(x)$ is the sum of the components of x, defining a hyperplane. We have thus several (promising) points $\bar{x}_{[k]}$ around which a careful search will be carried out.

To show the interest of this extra constraint, take again the previous example with $k = 1, 2, 3$, leading to three relaxed problems:

$$\begin{array}{lllll} \text{MKP}[1] & \rightarrow & \bar{z}_{[1]} = 12 & \text{et} & \bar{x}_{[1]} = (1, 0, 0, 0, 0) \\ \text{MKP}[2] & \rightarrow & \bar{z}_{[2]} = 24 & \text{et} & \bar{x}_{[2]} = (1, 1, 0, 0, 0) \\ \text{MKP}[3] & \rightarrow & \bar{z}_{[3]} = 25 & \text{et} & \bar{x}_{[3]} = (0, 0, 1, 1, 1) \end{array}$$

where $k = 3$ gives directly an optimal binary solution $x_{[3]} = x^{opt}$ without further search.

In general cases, search is necessary to explore around each fractional optimum and this is carried out by an effective TS algorithm (Section 5). In order not to go far away from each fractional optimum $\bar{x}_{[k]}$, we constraint TS to explore only the points $x \in \mathcal{S}$ such that $|x, \bar{x}_{[k]}|$, the distance between x and $\bar{x}_{[k]}$ is no greater than a given threshold δ_{max}[2].

To sum up, the proposed approach is composed of three steps:

1. determine interesting hyperplanes $\sigma(x) = k$;
2. run simplex to obtain the corresponding fractional optimum $\bar{x}_{[k]}$;
3. run Tabu Search around each $\bar{x}_{[k]}$, limited to a sphere of fixed radius.

4 Determine Hyperplanes $\sigma(x) = k$ and Simplex Phase

Given an n size instance of MKP01, it is obvious that the $n+1$ values $0 \leq k \leq n$ do not have the same interest regarding the optimum. Furthermore exploring all the hyperplanes $\sigma(x) = k$ would be too much time consuming. We propose, in this section, a simple way to compute good values of k.

Starting from a 0–1 lower bound z (*i.e.* a feasible solution obtained by a previous heuristic), the method consists in solving, with the *simplex* algorithm, the two following problems:

$$\text{MKP}\sigma min[z] \left\{ \begin{array}{l} \text{minimize } \sigma(x) \text{ s.t.} \\ A.x \leq b \text{ and } x \in [0\text{–}1]^n \text{ and} \\ c.x \geq (z+1) \end{array} \right.$$

Let σmin be the optimal value of this problem. A knapsack that holds fewer items than $k_{min} = \lceil \sigma min \rceil$ respects no longer the constraint $c.x \geq (z+1)$.

$$\text{MKP}\sigma max[z] \left\{ \begin{array}{l} \text{maximize } \sigma(x) \text{ s.t.} \\ A.x \leq b \text{ and } x \in [0\text{–}1]^n \text{ and} \\ c.x \geq (z+1) \end{array} \right.$$

Let σmax be the optimal value of this problem. It is not possible to hold more items than $k_{max} = \lfloor \sigma max \rfloor$ without violate one of the m constraints $A_i.x \leq b_i$.

[1] This hypothesis is confirmed by experimental results presented later. Of course, the hypothesis does not exclude the possibility that other areas may contain good solutions.

[2] This point is discussed more formally in section 5.2.

Consequently, local search will run only in the hyperplanes $\sigma(x) = k$ for which k is bounded by k_{min} and k_{max}. Hence we compute, using once again the *simplex* algorithm, the $(k_{max} - k_{min} + 1)$ relaxed MKP[k] which give us the continuous optima $\bar{x}_{[k]}$ which are used by the TS phase.

5 Tabu Search Phase

A comprehensive introduction of Tabu Search may be found in [Glover and Laguna, 1997]. We give briefly below some notations necessary for the understanding of our TS algorithm TS^{MKP}.

5.1 Notations

The following notations are specifically oriented to the MKP01:

- a **configuration** x is a binary vector with n components;
- the unconstrained **search space** $\mathcal{S}$ is defined to equal the set $\{0,1\}^n$, including both feasible and unfeasible configurations;
- a **move** consists in changing a small set of components of x giving x' and is denoted by $mvt(x, x')$;
- in this binary context, the flipped variables of a move can be identified by their indexes in the x vector: These indexes are the **attributes** of the move;
- the **neighborhood** of x, $\mathcal{N}(x)$, is the subset of configurations reachable from x in one move. In this binary context, a move from x to $x' \in \mathcal{N}$ can be identified without ambiguity by the attribute j if x' is obtained by flipping the j_{th} element of x. More generally, we use $mvt(i_1, i_2, ..., i_k)$ to denote the move $mvt(x, x')$ where $\forall j \in [1, k]\, x'_{i_j} = 1 - x_{i_j}$. Such a move is called a k_change.

5.2 Search Space Reduction

Regarding the knapsack constraints ($A.x \leq b$), $\mathcal{S} = \{0, 1\}^n$ is the completely relaxed search space. It is also the largest possible search space. We specify in this section a reduced search space $\mathcal{X} \subset \mathcal{S}$ which will be explored by our Tabu Search algorithm. To define this reduced space, we take the ideas discussed previously (Section 3):

1. limitation of $\mathcal{S}$ to a *sphere* of fixed radius around the point $\bar{x}_{[k]}$, the optimal solution of the relaxed MKP[k]. That is: $|x, \bar{x}_{[k]}| \leq \delta_{max}$;
2. limitation of the number of objects taken in the configurations x, that are visited by the tabu search algorithm, to the constant k (intersection of $\mathcal{S}$ and the hyperplane $\sigma(x) = k$).

For the first point, we use $\sum_{j=1}^{n} |x_j - x'_j|$ to define the distance $|x, x'|$ where x et x' may be binary or continuous. We use the following heuristic to estimate the maximal distance δ_{max} authorized from the starting point $\bar{x}_{[k]}$. Let $(1, 1, \cdots, 1, r_1, \cdots, r_q, 0, \cdots, 0)$ be the elements of the vector $\bar{x}_{[k]}$ sorted in decreasing order. r_j are fractional components of $\bar{x}_{[k]}$ and we have: $1 > r_1 \geq r_2 \geq \cdots \geq r_q > 0$. Let u be the number of the components having the value of 1 in $\bar{x}_{[k]}$. In the worst case, we select r_j items rather than the u components. With $\sigma(\bar{x}_{[k]}) = k$ that gives $\delta_{[k]} = 2 \times (u + q - k)$. If $u = k$, it follows $\delta_{[k]} = 0$ that corresponds to the case where $\bar{x}_{[k]}$ is a binary vector. Depending on the MKP01 instances, we choose $\delta_{max} \in [0.7 \times \delta_{[k]}, 2 \times \delta_{[k]}]$. Hence, each tabu process running around $\bar{x}_{[k]}$ has its own search space $\mathcal{X}_k$:

$$\mathcal{X}_k = \{x \in \{0,1\}^n \,|\, \sigma(x) = k \wedge |x, \bar{x}_{[k]}| \leq \delta_{max}\}$$

Note that all $\mathcal{X}_k$ are disjoint, this is a good feature for a distributed implementation of our approach.

Finally, in order to further limit TS to interesting areas of $\mathcal{X}_k$, we add a qualitative constraint on visited configurations:

$$c.x > z_{min}$$

where z_{min} is the best value of a feasible configuration found so far.

5.3 Neighborhood

A neighborhood which satisfies the constraint $\sigma(x) = k$ is the classical add/drop neighborhood: we remove an object from the current configuration and add another object to it at the same time. This neighborhood is used in this study. The set of neighboring configurations $\mathcal{N}(x)$ of a configuration x is thus defined by the formula:

$$\mathcal{N}(x) = \{x' \in \mathcal{X}_k / \, |x, x'| = 2\}$$

It is easy to see that this neighborhood is a special ase of 2_change (cf. sect. 5.1). We use in the following sections indifferently $mvt(x, x')$ or $mvt(i, j)$ with $x'_i = 1 - x_i$ and $x'_j = 1 - x_j$. As TS evaluates the whole neighborhood for a move, we have a time complexity of $O((n - k) \times k)$ for each iteration of TS^{MPK}.

5.4 Tabu List Management

The reverse elimination method (REM), introduced by Fred Glover [Glover, 1990], leads to an exact tabu status (*i.e.* $mvt(x, x')$ tabu $\Leftrightarrow$ x' has been visited). It consists in storing in a *running list* the attributes (pair of components) of all completed moves. Telling if a move is forbidden or not needs to trace back the *running list*. Doing so, one builds another list, the so-called residual cancellation sequence (RCS) in which attributes are either added, if they are not yet in the RCS, or dropped otherwise. The condition $RCS = \emptyset$ corresponds to a move leading to a yet visited configuration. For more details on this method see [Dammeyer and Voß, 1993; Glover, 1990]. REM has a time complexity $O(iter^2)$ ($iter$ is the current value of the move counter).

For our specific 2_change move, we need only one trace of the *running list*. Each time we meet $|RCS| = 2$, the move with RCS_0 and RCS_1 attributes is said tabu. Let $iter$ be the move counter. The following algorithm updates the tabu status of the whole neighborhood of a configuration x and corresponds to the equivalence

$$iter = tabu[i][j] \Longleftrightarrow mvt(i, j)\, tabu$$

Algorithm 1: UPDATE_TABU

$i \leftarrow erl$ % end of *running list*
repeat
  $i \leftarrow i - 1$
  $j \leftarrow$ *running list*$[i]$
  if $j \in RCS$ **then**
    $RCS \leftarrow RCS \ominus j$
  else
    $RCS \leftarrow RCS \oplus j$
    if $|RCS| = 2$ **then**
      $tabu[RCS_0][RCS_1] \leftarrow iter$
      $tabu[RCS_1][RCS_0] \leftarrow iter$
until $i = 0$

This algorithm traces back the *running list* table from its $erl - 1$ entry to its 0 entry.

5.5 Evaluation Function

We use a two components evaluation function to assess the configurations $x \in S$: $v_b(x) = \sum_{i\,|\,a_i x > b_i} (a_i.x - b_i)$ and $z(x) = c.x$. To make a move from x, we take among $x' \in \mathcal{N}(x)$ the configuration x' defined by the following relation:

$$x' \in \mathcal{N}(x) \text{ such that } \begin{cases} \forall\, x'' \in \mathcal{N}(x) \\ v_b(x') < v_b(x'') \text{ or} \\ v_b(x') = v_b(x'') \text{ and } c.x' \geq c.x'' \end{cases}$$

Random choice is used to break ties.

Each time $v_b(x) = 0$ the *running list* is reset and z_{min} is updated. Hence the tabu algorithm explores areas of the search space where $z > z_{min}$ trying to reach feasibility.

5.6 Tabu Algorithm

Based on the above considerations, our TS algorithm for MKP01 (TS^{MPK}) works on a series of problems like:

$$\text{find } x \in \mathcal{S} = \{0,1\}^n \text{ such that}$$
$$A.x \leq b \wedge \textstyle\sum_1^n x_i = k \wedge |x, \bar{x}_{[k]}| \leq \delta_{max} \wedge c.x > z_{min}$$

where z_{min} is a strictly increasing sequence. Let $|R.L.|$ be the *running list* size. Considering the last feature of our algorithm given just above (sect. 5.5), $|R.L.|$ is twice the maximum number of iterations without improvement of the cost function. The starting configuration x_{init} is built with the following simple greedy heuristic: Choose the k items $\{i_1, \cdots, i_k\}$, which have the largest $\bar{x}_{[k]i_j}$ value.

6 Computational Results

All the procedures of TS^{MPK} have been coded in C language. Our algorithm has been executed on different kind of *CPU* (up to 20 distributed computers like PII350, PIII500, ULTRASPARC5 and 30). For all the instances solved below, we run TS^{MPK} with 10 random seeds (0..9) of the standard *srand()* C function.

We have first tested our approach on the 56 classical problems used in [Aboudi and Jörnsten, 1994; Balas and Martin, 1980; Chu and Beasley, 1998; Dammeyer and Voß, 1993; Drexl, 1988; Fréville and Plateau, 1993; Glover and Kochenberger, 1996; Shih, 1979; Toyoda, 1975]. The size of these problems varies from n=6 to 105 items and from m=2 to 30 constraints. These instances are easy to solve for state-of-the-art algorithms. Indeed, our approach finds the optimal value

Algorithm 2: TS^{MKP}

$iter \leftarrow 0$ % Move counter
$(erl, tabu[][]) \leftarrow (0, (-1, ..., -1))$
$x \leftarrow x_{init}$
if $v_b(x) = 0$ **then**
  $z_{min} \leftarrow c.x$; $x^* \leftarrow x$
else
  $z_{min} \leftarrow 0$; $x^* \leftarrow (0)$
repeat
  $(v_{min}, z_{max}) \leftarrow (\infty, -\infty)$
  for $1 \leq i < j \leq n$ **do**
    if $tabu[i][j] \neq iter$ **then**
      $(x_i, x_j) \leftarrow (1 - x_i, 1 - x_j)$ %$mvt(i,j)$ evaluation
      if $(|x, \bar{x}_{[k]}| \leq \delta_{max}) \wedge (c.x > z_{min})$ **then**
        if $(v_b(x) < v_{min}) \vee (v_b(x) = v_{min} \wedge c.x > z_{max})$ **then**
          $(i', j') \leftarrow (i, j)$
          $(v_{min},\ z_{max}) \leftarrow (v_b(x),\ c.x)$
        $(x_i, x_j) \leftarrow (1 - x_i, 1 - x_j)$ % Restore old values
  if $v_{min} \neq \infty$ **then**
    $(x_{i'}, x_{j'}) \leftarrow (1 - x_{i'}, 1 - x_{j'})$ % Complete move
    if $v_{min} = 0$ **then**
      $erl \leftarrow 0$ %Reset the *running list*
      $z_{min} \leftarrow c.x$
      $x^* \leftarrow x$
    else
      $iter \leftarrow iter + 1$
      running list $\leftarrow$ *running list* $\oplus\ i' \oplus\ j'$
      $erl \leftarrow erl + 2$
      UPDATE_TABU
until $v_{min} = \infty \vee erl \geq |R.L.|$

(known for all these instances) in an average time of 1 second (details are thus omitted). The *running list* size ($|R.L.|$) is fixed to 4000.

For the next two sets of problems, the *running list* size ($|R.L.|$) is fixed to 100000. Hence the maximum number of moves without improvement is 50000.

The second set of tested instances is constituted of the last seven (also the largest ones with n =100 to 500 items, m = 15 to 25 constraints) of 24 benchmarks proposed by Glover and Kochenberger [Glover and Kochenberger, 1996]. These instances are known to be particularly difficult to solve for *Branch & Bound* algorithms. Table **1** shows a comparison between our results (columns 4 to 7) and the best known ones reported in [Hanafi and Fréville, 1998] (column T_{HF}).

GK	$n \times m$	T_{HF}	z^*	k^*	$iter^*$	$sec.^*$	$[k]$
18	100×25	4524	**4528**	61	3683	10	58..64
19	100×25	3866	**3869**	51	3144	9	49..55
20	100×25	5177	**5180**	70	2080	5	61..74
21	100×25	3195	**3200**	42	1465	4	40..46
22	100×25	2521	**2523**	34	512	2	31..37
23	200×15	9231	**9235**	123	16976	131	119..126
24	500×25	9062	**9070**	119	9210	268	116..125

Table 1: *Comparative results on the 7 largest* GK *pb.*

Column k^* shows the number of items of the best solution x^* with cost z^* found by TS^{MKP}. From the table, we observe that all the results are improved. Columns $iter^*$ and $sec.^*$ give the number of moves and the time elapsed to reach x^*. We know that the algorithm runs 50000 moves after $iter^*$. The search process takes thus an average of 380 seconds for the test problems GK18...GK22, 600 seconds for GK23 and 1100 seconds for GK24. Last column gives the k hyperplane's values visited by TS^{MKP}.

The third set of test problems concerns the 30 largest

benchmarks ($n = 500$ items, $m = 30$ constraints) of OR-Library[3], proposed recently by Chu and Beasley [Chu and Beasley, 1998].

CB	AG_{CB}	z^*	k^*	$iter^*$	$sec.^*$	$[k]$
30.500.0	115868	**115950**	130	17841	397	128..133
30.500.1	114667	**114810**	128	104866	2264	125..131
30.500.2	116661	**116683**	128	73590	1203	125..132
30.500.3	115237	**115301**	128	71820	1587	125..131
30.500.4	116353	**116435**	127	75909	1784	125..133
30.500.5	115604	**115694**	131	33391	684	128..134
30.500.6	113952	**114003**	128	107994	2851	126..132
30.500.7	114199	**114213**	129	87593	1503	125..132
30.500.8	115247	**115288**	127	75243	1495	125..132
30.500.9	116947	**117055**	129	39044	869	125..132
30.500.10	217995	**218068**	251	6828	116	249..254
30.500.11	214534	**214562**	251	89201	2478	249..254
30.500.12	215854	**215903**	250	60074	1311	248..253
30.500.13	217836	**217910**	251	50732	1121	249..255
30.500.14	215566	**215596**	251	62524	1262	248..254
30.500.15	215762	**215842**	253	34201	633	250..257
30.500.16	215772	**215838**	252	54476	1003	250..256
30.500.17	216336	**216419**	253	40683	947	250..257
30.500.18	217290	**217305**	253	64489	1475	250..257
30.500.19	214624	**214671**	252	18531	368	250..256
30.500.20	301627	**301643**	375	1298	17	373..378
30.500.21	299985	**300055**	374	78278	1532	372..379
30.500.22	304995	**305028**	375	64926	1161	373..379
30.500.23	301935	**302004**	375	26901	1110	373..379
30.500.24	304404	**304411**	376	20483	333	374..379
30.500.25	296894	**296961**	374	31403	462	371..377
30.500.26	303233	**303328**	373	43398	757	372..377
30.500.27	306944	**306999**	376	33810	1366	374..380
30.500.28	303057	**303080**	374	17647	350	374..380
30.500.29	300460	**300532**	376	6948	150	373..379

Table 2: *Comparative results on the 30 largest* CB *pb.*

Table **2** compares our results with those reported in [Chu and Beasley, 1998] (AG_{CB}), which are among the best results for these instances. From the table, we see that our approach improves significantly on all these results. The average time of the 50000 last iterations is equal to 1200 seconds for these instances.

Column "$[k]$" shows that an average of seven hyperplanes are scanned by the TS^{MKP} procedure for a given instance.

These results can be further improved by giving more *CPU* time (iterations) to our algorithm. For example, with $|R.L.| = 300000$, TS^{MKP} finds $z^* = 115991$ after 6000 seconds for the instance CB30.500.0.

The Chu and Beasley *benchmark* contains 90 instances with 500 variables: 30 instances with m=5 constraints, 30 with $m = 10$ and 30 with $m = 30$ (results detailed just above). Each set of 30 instances is divided into 3 series with $\alpha = b_i / \sum_{j=1}^{n} A_{ij} = 1/4$, $\alpha = 1/2$ and $\alpha = 3/4$. Table **3** compares, for each subset of 10 instances, the averages of the best results obtained by AG_{CB}, those obtained more recently by Osorio, Glover and Hammer [Osorio *et al.*, 2000] (columns 4 and 5) and those by TS^{MPK} (column 6). The new algorithm of [Osorio *et al.*, 2000] uses advanced techniques such as cutting and surrogate constraint analysis (see column *Fix+Cuts* for results). We reproduce also from [Osorio *et al.*, 2000], in column $CPLEX$, the best values obtained by the *MIP solver CPLEX v6.5.2* alone.

[3] Available at `http://mscmga.ms.ic.ac.uk`.

m	α	AG_{CB}	*Fix+Cuts*	*CPLEX*	z^*	gap
5	1/4	120616	120610	120619	**120623**	0.08%
	1/2	219503	219504	219506	**219507**	0.04%
	3/4	302355	**302361**	302358	302360	0.02%
10	1/4	118566	118584	118597	**118600**	0.20%
	1/2	217275	217297	217290	**217298**	0.09%
	3/4	302556	302562	302573	**302575**	0.07%
30	1/4	115470	115520	115497	**115547**	0.55%
	1/2	216187	216180	216151	**216211**	0.24%
	3/4	302353	302373	302366	**302404**	0.15%

Table 3: *Average performance over the 90 largest* CB *pb.*

The column gap indicates the average gap values in percentage between the continuous relaxed optimum and the best cost value found: $(\bar{z} - z^*)/\bar{z}$. $Fix+cuts$ and $CPLEX$ algorithms were stopped after 3 hours of computing or when the tree size memory of 250M bytes was exceeded. Our best results were obtained with $|R.L.| = 300000$ and the algorithm never requires more than 4M bytes of memory. Except for the instances with $m = 5$ and $\alpha = 3/4$ our approach outperforms all the other algorithms.

To finish the presentation on Chu and Beasley benchmarks, Table **4** and **5** show the best values obtained by our algorithm on the 30 CB5.500 and the 30 CB10.500 instances.

XX	z^*	k^*	XX	z^*	k^*	XX	z^*	k^*
0	120134	146	10	218428	267	20	295828	383
1	117864	148	11	221191	265	21	308083	383
2	121112	145	12	217534	264	22	299796	385
3	120804	149	13	223558	264	23	306478	385
4	122319	147	14	218966	267	24	300342	385
5	122024	153	15	220530	262	25	302561	384
6	119127	145	16	219989	266	26	301329	385
7	120568	150	17	218194	266	27	306454	383
8	121575	148	18	216976	262	28	302822	382
9	120707	151	19	219704	267	29	299904	379

Table 4: *Best values for* CB5.500.XX

XX	z^*	k^*	XX	z^*	k^*	XX	z^*	k^*
0	117779	134	10	217343	256	20	304351	378
1	119190	134	11	219036	259	21	302333	380
2	119194	135	12	217797	256	22	302408	379
3	118813	137	13	216836	258	23	300757	378
4	116462	134	14	213859	256	24	304344	381
5	119504	137	15	215034	257	25	301754	375
6	119782	139	16	217903	261	26	304949	378
7	118307	135	17	219965	256	27	296441	379
8	117781	136	18	214341	258	28	301331	379
9	119186	138	19	220865	255	29	307078	378

Table 5: *Best values for* CB10.500.XX

To conclude this section, we present our results on the 11 latest instances proposed very recently by Glover and Kochenberger (available at: `http://hces.bus.olemiss.edu/tools.html`.) These benchmarks contain up to n=2500 items and m=100 constraints, thus are very large. Table **6** compares our results (columns 4 and 5) and the best known results taken from the above web site. Once again, our approach gives improved solutions for 9 out

of 11 instances. Let us mention that the experimentation on these instances showed some limits of our approach regarding the computing time for solving some very large instances ($n > 1000$). Indeed, given the size of our neighborhood ($k \times (n-k)$, see Section 5.3), up to 3 days were needed to get the reported values.

MK_GK	$n \times m$	z^{GK}	z^*	k^*	gap
01	100×15	3766	3766	52	0.26%
02	100×25	3958	3958	50	0.46%
03	150×25	5650	**5656**	78	0.26%
04	150×50	5764	**5767**	78	0.46%
05	200×25	7557	**7560**	104	0.23%
06	200×50	7672	**7677**	99	0.34%
07	500×25	19215	**19220**	259	0.06%
08	500×50	18801	**18806**	252	0.13%
09	1500×25	58085	**58087**	773	0.02%
10	1500×50	57292	**57295**	770	0.04%
11	2500×100	95231	**95237**	1271	0.04%

Table 6: *Comparison on the lastest 11* MK_GK *pb.*

7 Conclusion

In this paper, we have presented a hybrid approach for tackling the NP-hard 0–1 multidimensional knapsack problem (MKP01). The proposed approach combines linear progamming and Tabu Search. The basic idea consists in obtaining "promising" continuous optima and then using TS to explore carefully and efficiently binary areas close to these continuous optima. This is carried out by introducing several additional constraints:

1. hyperplane constraint: $\sum_1^n x_i = k \in \mathbb{N}$;
2. geometrical constraint: $|x, \bar{x}_{[k]}| \leq \delta_{max}$;
3. qualitative constraint: $c.x > z_{min}$.

Our Tabu Search algorithm integrates also some advanced features such as an efficient implementation of the reverse elimination method for a dynamic tabu list management in the context of a special *2-change* move.

This hybrid approach has been tested on more than 100 state-of-the-art benchmark instances, and has led to improved solutions for most of the tested instances.

One inconvenience of the proposed approach is its computing time for very large instances ($n > 1000$ items). This is partially due to the time complexity of the neighborhood used. Despite of this, this study constitutes a promising starting point for designing more efficient algorithms for the MKP01. Some possibilities are: 1) to develop a partial evaluation of relaxed knapsack constraints; 2) to study more precisely the relationship between δ_{max} and z^{opt} for a given instance; 3) to find a good crossover operator and a cooperative distributed implementation of the TS^{MKP} algorithm.

Finally, we hope the basic idea behind the proposed approach may be explored to tackle other NP-hard problems.

Acknowledgement: We would like to thank the reviewers of the paper for their useful comments.

References

[Aboudi and Jörnsten, 1994] R. Aboudi and K. Jörnsten. Tabu Search for General Zero-One Integer Programs using the Pivot and Complement Heuristic. *ORSA Journal of Computing*, 6(1):82–93, 1994.

[Balas and Martin, 1980] E. Balas and C.H. Martin. Pivot and a Complement Heuristic for 0-1 Programming. *Management Science*, 26:86–96, 1980.

[Chu and Beasley, 1998] P.C. Chu and J.E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristic*, 4:63–86, 1998.

[Dammeyer and Voß, 1993] F. Dammeyer and S. Voß. Dynamic tabu list management using the reverse elimination method. *Annals of Operations Research*, 41:31–46, 1993.

[Drexl, 1988] A. Drexl. A simulated annealing approach to the multiconstraint zero-one knapsack problem. *Computing*, 40:1–8, 1988.

[Fréville and Plateau, 1993] A. Fréville and G. Plateau. Sac à dos multidimensionnel en variable 0-1 : encadrement de la somme des variables à l'optimum. *Recherche Opérationnelle*, 27(2):169–187, 1993.

[Gavish and Pirkul, 1982] B. Gavish and H. Pirkul. Allocation of data bases and processors in a distributed computting system. *Management of Distributed Data Processing*, 31:215–231, 1982.

[Gavish and Pirkul, 1985] B. Gavish and H. Pirkul. Efficient algorithms for solving multiconstraint zero-one knapsack problems to optimality. *Mathematical Programming*, 31:78–105, 1985.

[Glover and Kochenberger, 1996] F. Glover and G.A. Kochenberger. Critical event tabu search for multidimensional knapsack problems. In I.H. Osman J.P. Kelly, editor, *Metaheuristics: The Theory and Applications*, pages 407–427. Kluwer Academic Publishers, 1996.

[Glover and Laguna, 1997] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[Glover, 1990] F. Glover. Tabu search. *ORSA Journal of Computing*, 2:4–32, 1990.

[Hanafi and Fréville, 1998] S. Hanafi and A. Fréville. An efficient tabu search approach for the 0-1 multidimensional knapsack problem. *European Journal of Operational Research*, 106:659–675, 1998.

[Osorio *et al.*, 2000] M.A. Osorio, F. Glover, and Peter Hammer. Cutting and surrogate constraint analysis for improved multidimensional knapsack solutions. Technical report, Hearin Center for Enterprise Science. Report HCES-08-00, 2000.

[Shih, 1979] W. Shih. A branch & bound method for the multiconstraint zero-one knapsack problem. *Journal of the Operational Research Society*, 30:369–378, 1979.

[Toyoda, 1975] Y. Toyoda. A simplified algorithm for obtaining approximate solutions to zero-one programming problem. *Management Science*, 21(12):1417–1427, 1975.

The Exponentiated Subgradient Algorithm for Heuristic Boolean Programming

Dale Schuurmans and **Finnegan Southey**
Department of Computer Science
University of Waterloo
{dale,fdjsouth}@cs.uwaterloo.ca

Robert C. Holte*
Department of Computing Science
University of Alberta
holte@cs.ualberta.ca

Abstract

Boolean linear programs (BLPs) are ubiquitous in AI. Satisfiability testing, planning with resource constraints, and winner determination in combinatorial auctions are all examples of this type of problem. Although increasingly well-informed by work in OR, current AI research has tended to focus on specialized algorithms for each type of BLP task and has only loosely patterned new algorithms on effective methods from other tasks. In this paper we introduce a single general-purpose local search procedure that can be simultaneously applied to the entire range of BLP problems, without modification. Although one might suspect that a general-purpose algorithm might not perform as well as specialized algorithms, we find that this is not the case here. Our experiments show that our generic algorithm simultaneously achieves performance comparable with the state of the art in satisfiability search and winner determination in combinatorial auctions—two very different BLP problems. Our algorithm is simple, and combines an old idea from OR with recent ideas from AI.

1 Introduction

A Boolean linear program is a constrained optimization problem where one must choose a set of binary assignments to variables $x_1, ..., x_n$ to satisfy a given set of m linear inequalities $\mathbf{c}_1 \cdot \mathbf{x} \leq b_1, ..., \mathbf{c}_m \cdot \mathbf{x} \leq b_m$ while simultaneously optimizing a linear side objective $\mathbf{a} \cdot \mathbf{x}$. Specifically, we consider BLP problems of the canonical form

$$\min_{\mathbf{x} \in \{-1,+1\}^n} \mathbf{a} \cdot \mathbf{x} \quad \text{subject to} \quad C\mathbf{x} \leq \mathbf{b} \qquad (1)$$

Clearly this is just a special case of integer linear programming (ILP) commonly referred to as 0-1 integer programming.[1] Many important problems in AI can be naturally expressed as BLP problems; for example: finding a satisfying truth assignment for CNF propositional formulae (SAT) [Kautz and Selman, 1996], winner determination in combinatorial auctions (CA) [Sandholm, 1999; Fujishima *et al.*, 1999], planning with resource constraints [Kautz and Walser, 1999; Vossen *et al.*, 1999], and scheduling and configuration problems [Walser, 1999; Lau *et al.*, 2000]. The two specific problems we will investigate in detail below are SAT and CA.

In general, BLP problems are hard in the worst case (NP-hard as optimization problems, NP-complete as decision problems). Nevertheless, they remain important problems and extensive research has been invested in improving the exponential running times of algorithms that guarantee optimal answers, developing heuristic methods that approximate optimal answers, and identifying tractable subcases and efficient algorithms for these subcases. There is a fundamental distinction between *complete* methods, which are guaranteed to terminate and produce an optimal solution to the problem, and *incomplete* methods, which make no such guarantees but nevertheless often produce good answers reasonably quickly. Most complete methods conduct a systematic search through the variable assignment space and use pruning rules and ordering heuristics to reduce the number of assignments visited while preserving correctness. Incomplete methods on the other hand typically employ local search strategies that investigate a small neighborhood of a current variable assignment (by flipping one or a small number of assignments) and take greedy steps by evaluating neighbors under a heuristic evaluation function. Although complete methods might appear to be more principled, incomplete search methods have proven to be surprisingly effective in many domains, and have led to significant progress in some fields of research (most notably on SAT problems [Kautz and Selman, 1996] but more recently on CA problems as well [Hoos and Boutilier, 2000]).

In this paper we investigate incomplete local search methods for general BLP problems. Although impressive, most of the recent breakthroughs in incomplete local search have been achieved by tailoring methods to a reduced class of BLP problems. A good illustration of this point is the CA system of Hoos and Boutilier [2000] (Casanova) which is based on the SAT system of [Hoos, 1999; McAllester *et al.*, 1997] (Novelty+), but is not the same algorithm (neither of these algorithms can be directly applied to the opposing problem). Another example is the WSAT(OIP) system of Walser [1999] which is an extension of WSAT [Selman *et al.*, 1994] and achieves impressive results on general ILP problems, but nev-

*Work performed while at the University of Ottawa.

[1] Although one could alternatively restrict the choice of values to $\{0, 1\}$ we often find it more convenient to use $\{-1, +1\}$.

ertheless requires soft constraints to be manually created to replace a given optimization objective.

Our main contribution is the following: Although a variety of research communities in AI have investigated BLP problems and developed effective methods for certain cases, the current level of specialization might not be yielding the benefits that one might presume. To support this observation we introduce a generic local search algorithm that when applied to specialized forms of BLP (specifically SAT and CA) achieves performance that competes with the state of the art on these problems. Our algorithm is based on combining a simple idea from OR (subgradient optimization for Lagrangian relaxation) with a recent idea from AI, specifically from machine learning theory (multiplicative updates and exponentiated gradients). The conclusion we draw is that research on general purpose methods might still prove fruitful, and that known ideas in OR might still yield additional benefits in AI problems beyond those already acknowledged.

2 Background

Research on constrained optimization in AI has become increasingly well informed by extensive foundational work in OR [Gomes, 2000]. OR has investigated the specific task of ILP in greater depth than AI, and has tended to place more emphasis on developing general purpose methods applicable across the entire range of ILP problems. Complete methods in OR typically employ techniques such as branch and bound (using linear programming or Lagrangian relaxation), cutting planes, and branch and cut to prune away large portions of the assignment space in a systematic search [Martin, 1999]. This research has yielded sophisticated and highly optimized ILP solvers, such as the commercial CPLEX system, which although general purpose, performs very well on the specialized problems investigated in AI. In fact, it is still often unclear whether specialized AI algorithms perform better [Andersson *et al.*, 2000].

Perhaps less well known to AI researchers is that beyond systematic search with branch and bound, the OR literature also contains a significant body of work on *incomplete* (heuristic) local search methods for approximately solving ILP problems; see for example [Magazine and Oguz, 1984; Larsson *et al.*, 1996]. The simplest and most widespread of these ideas is to use *subgradient optimization* (which we describe below).

Our algorithm arises from the observation that the most recent strategies developed for the SAT problem have begun to use an analog of subgradient optimization as their core search strategy; in particular the DLM system of [Wu and Wah, 2000] and the SDF system of [Schuurmans and Southey, 2000] (see also [Thornton and Sattar, 1999; Frank, 1997; Davenport *et al.*, 1994]). Interestingly, these are among the most effective methods for finding satisfying assignments for CNF formulae, and yet they appear to be recapitulating a forty year old idea in OR going back to [Everett, 1963]. Below we show that, indeed, a straightforward subgradient optimization approach (with just three basic improvements from AI) yields a state of the art SAT search strategy that competes with DLM (arguably the fastest SAT search procedure currently known).

Interestingly, the method we develop is not specialized to SAT problems in any way. In fact, the algorithm is a general purpose BLP search technique that can in principle be applied to any problem in this class. To demonstrate this point we apply the method *without modification* (beyond parameter setting) to CA problems and find that the method still performs well relative to the state of the art (although somewhat less impressively than on SAT problems). In this case we also find that the commercial CPLEX system performs very well and is perhaps the best of the methods that we investigated. Our results give us renewed interest in investigating general purpose algorithms for BLP that combine well understood methods from OR with recent ideas from AI.

3 The exponentiated subgradient algorithm

To explain our approach we first need to recap some basic concepts from constrained optimization theory. Consider the canonical BLP (1). For any constrained optimization problem of this form the *Lagrangian* is defined to be

$$L(\mathbf{x}, \mathbf{y}) \quad = \quad \mathbf{a} \cdot \mathbf{x} + \sum_{i=1}^{m} y_i (\mathbf{c}_i \cdot \mathbf{x} - b_i) \tag{2}$$

where $\mathbf{c}_i$ is the ith row vector of the constraint matrix C and y_i is the real valued Lagrange multiplier associated with constraint c_i. One can think of the multipliers y_i simply as weights on the constraint violations $v_i \triangleq \mathbf{c}_i \cdot \mathbf{x} - b_i$. Thus the Lagrangian can be thought of as a penalized objective function where for a given vector of constraint violation weights one could imagine minimizing the penalized objective $L(\mathbf{x}, \mathbf{y})$. In this way, the Lagrangian turns a constrained optimization problem into an unconstrained optimization problem. In fact, the solutions of these unconstrained optimizations are used to define the *dual function*

$$D(\mathbf{y}) \quad = \quad \min_{\mathbf{x} \in \{-1,+1\}^n} L(\mathbf{x}, \mathbf{y}) \tag{3}$$

The *dual problem* for (1) is then defined to be

$$\max_{\mathbf{y}} D(\mathbf{y}) \quad \text{subject to} \quad \mathbf{y} \geq \mathbf{0} \tag{4}$$

Let D^* denote the maximum value of (4) and let P^* denote the minimum value of (1). Note that these are all just definitions. The reason that (4) can be considered to be a dual to (1) is given by the *weak duality theorem*, which asserts that $D^* \leq P^*$ [Bertsekas, 1995; Martin, 1999] (see Appendix A). Therefore, solving $\min_{\mathbf{x} \in \{-1,+1\}^n} L(\mathbf{x}, \mathbf{y})$ for any Lagrange multiplier vector $\mathbf{y} \geq \mathbf{0}$ gives a *lower bound* on the optimum value of the original problem (1). The best achievable lower bound is achieved by solving the dual problem of maximizing $D(\mathbf{y})$ subject to $\mathbf{y} \geq \mathbf{0}$, yielding the optimal value D^*. The difference $P^* - D^*$ is called the *duality gap*. A fundamental theorem asserts that there is no duality gap for linear (or convex) programs with real valued variables [Bertsekas, 1995], so in principle these problems can be solved by dual methods alone. However, this is no longer the case once the variables are constrained to be integer or $\{-1, +1\}$ valued.

Although the dual problem cannot be used to directly solve (1) because of the existence of a possible duality gap, obtaining lower bounds on the optimal achievable value can still

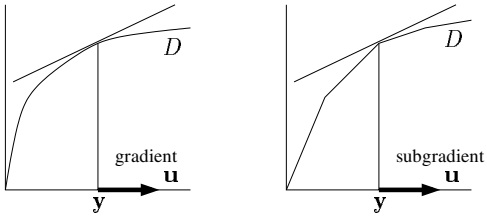

Figure 1: Subgradients generalize gradients to nondifferentiable functions. Note that the subgradient is not uniquely defined at a nondifferentiable point.

prove very useful in various search strategies, ranging from branch and bound to local search. Obviously there is a natural desire to maximize $D(\mathbf{y})$ by searching through Lagrange multiplier vectors for larger values. A natural way to proceed would be to start with a vector $\mathbf{y}^{(0)}$, solve the unconstrained minimization problem (3), and then update $\mathbf{y}^{(0)}$ to obtain a better weight vector $\mathbf{y}^{(1)}$, and so on. The issue is knowing how to update the weight vector $\mathbf{y}^{(0)}$. This question is complicated by the fact that $D(\mathbf{y})$ is typically nondifferentiable, which leads to the last technical development we will consider: *subgradient optimization*.

Since $D(\mathbf{y})$ is always known to be concave [Bertsekas, 1995], a *subgradient* of D (at a point $\mathbf{y}$) is given by any direction vector $\mathbf{u}$ such that $D(\mathbf{z}) \leq D(\mathbf{y}) + (\mathbf{z} - \mathbf{y})^\top \mathbf{u}$ for all $\mathbf{z} \in I\!R^m$. (Intuitively, a subgradient vector $\mathbf{u}$ gives the increasing direction of a plane that sits above D but touches D at $\mathbf{y}$, and hence can serve as a plausible search direction if one wishes to increase D; see Figure 1.) Therefore, to update $\mathbf{y}$, all we need to do is find a subgradient direction. Here we can exploit an extremely useful fact: after minimizing $L(\mathbf{x}, \mathbf{y})$ to obtain $\mathbf{x_y} = \arg\min_{\mathbf{x}\in\{-1,+1\}^n} \mathbf{a}\cdot\mathbf{x} + \mathbf{y}^\top(C\mathbf{x} - \mathbf{b})$ *the resulting vector of residual constraint violation values* $\mathbf{v_y} = C\mathbf{x_y} - \mathbf{b}$ *is always a subgradient of* D *at* $\mathbf{y}$ [Bertsekas, 1995; Martin, 1999] (see Appendix B). So, in the end, despite the overbearing terminology, subgradient optimization is a fundamentally simple procedure:

Subgradient optimization: To improve the lower bound $D(\mathbf{y})$ on the optimal solution of the original problem, take a given vector of Lagrange multipliers $\mathbf{y}$, solve for a primal vector $\mathbf{x} \in \{-1,+1\}^n$ that minimizes the Lagrangian $L(\mathbf{x}, \mathbf{y})$, and update $\mathbf{y}$ by adding a proportional amount of the residual constraint violations $\mathbf{v_y} = C\mathbf{x_y} - \mathbf{b}$ to $\mathbf{y}$ (maintaining $\mathbf{y} \geq \mathbf{0}$); i.e. use the rule $\mathbf{y}' = \mathbf{y} + \alpha\mathbf{v_y}$. Note that this has the intuitive effect of increasing the weights on the violated constraints while decreasing weights on satisfied constraints. Although this update is not guaranteed to increase the value of $D(\mathbf{y})$ at every iteration, it is guaranteed to move $\mathbf{y}$ closer to $\mathbf{y}^* = \arg\max_{\mathbf{y}\geq\mathbf{0}} D(\mathbf{y})$ for sufficiently small step-sizes α [Bertsekas, 1995].

These ideas go back at least to [Everett, 1963], and have been applied with great success by Held and Karp [1970] and many since. Typically, subgradient optimization has been used as a technique for generating good lower bounds for branch and bound methods (where it is known as *Lagrangian relaxation* [Fisher, 1981]). However, subgradient optimization can also be used as a heuristic search method for approximately solving (1) in a very straightforward way: at each iteration simply check if $\mathbf{x_y}$ is feasible (i.e. satisfies $C\mathbf{x_y} \leq \mathbf{b}$), and, if so, report $\mathbf{a}\cdot\mathbf{x_y}$ as a feasible objective value (keeping it if it is the best value reported so far); see for example [Magazine and Oguz, 1984; Larsson *et al.*, 1996].

Interestingly, in the last few years this procedure has been inadvertently rediscovered in the AI literature. Specifically, in the field of incomplete local search methods for SAT, clause weighting schemes turn out to be using a form of subgradient optimization as their main control loop. These procedures work by fixing a profile of clause weights (Lagrange multipliers), greedily searching through variable assignment space for an assignment that minimizes a weighted score of clause violations (the Lagrangian), and then updating the clause weights by increasing them on unsatisfied clauses—all of which comprises a single subgradient optimization step. However, there are subtle differences between recent SAT procedures and the basic subgradient optimization approach outlined above. The DLM system of [Wu and Wah, 2000] explicitly uses a Lagrangian, but the multiplier updates follow a complex system of ad hoc calculations.[2] The SDF system of [Schuurmans and Southey, 2000] is simpler (albeit slower), but includes several details of uncertain significance. Nevertheless, the simplicity of this latter approach offers useful hypotheses for our current investigation.

Given the clear connection between recent SAT procedures and the traditional subgradient optimization technique in OR, we conduct a study of the deviations from the basic OR method so that we can identify the deviations which are truly beneficial. Our intent is to validate (or refute) the significance of some of the most recent ideas in the AI SAT literature.

Linear versus nonlinear penalties: One difference between recent SAT methods and subgradient optimization is that the SAT methods only penalize constraints with positive violations (by increasing the weight on these constraints), whereas standard subgradient optimization also rewards satisfied constraints (by reducing their weight proportionally to the degree of negative violation). To express this difference, consider an augmented Lagrangian which extends (2) by introducing a *penalty function* θ on constraint violations

$$L_\theta(\mathbf{x}, \mathbf{y}) \quad = \quad \mathbf{a}\cdot\mathbf{x} + \sum_{i=1}^{m} y_i\ \theta(\mathbf{c}_i\cdot\mathbf{x} - b_i)$$

The penalty functions we consider are the traditional linear penalty $\theta(v) = v$ and the "hinge" penalty

$$\theta(v) \quad = \quad \begin{cases} -\frac{1}{2} & \text{if } v \leq 0 \\ v - \frac{1}{2} & \text{if } v > 0 \end{cases}$$

(Note that v is an integer.) DLM and SDF can be interpreted as implicitly using a hinge penalty for dual updates on SAT problems. (However, SDF uses a different penalty for its primal search.) Intuitively, the hinge penalty has advantages because it does not favor increasing the satisfaction level of

[2] The Lagrangian used in [Wu and Wah, 2000] is also quite different from (2) because it includes a redundant copy of the constraint violations in the minimization objective. This prevents their Lagrangian from being easily generalized beyond SAT.

satisfied constraints above reducing the violations of unsatisfied constraints. Below we observe that the traditional linear penalty leads to poor performance on constraint satisfaction tasks and the AI methods have an advantage in this respect.

Unfortunately, the consequence of choosing a nonlinear penalty function is that finding a variable assignment $\mathbf{x}$ which minimizes $L_\theta(\mathbf{x}, \mathbf{y})$ is no longer tractable. To cope with this difficulty AI SAT solvers replace the global optimization process with a greedy local search (augmented with randomization). Therefore, they only follow a *local* subgradient direction in $\mathbf{y}$ at each update. However, despite the locally optimal nature of the dual updates, the hinge penalty appears to retain an advantage over the linear penalty.

Multiplicative versus additive updates: The SDF procedure updates $\mathbf{y}$ multiplicatively rather than additively, in an analogy to the work on multiplicative updates in machine learning theory [Kivinen and Warmuth, 1997]. A multiplicative update is naturally interpreted as following an exponentiated version of the subgradient; that is, instead of using the traditional additive update $\mathbf{y}' = \mathbf{y} + \alpha\boldsymbol{\theta}(\mathbf{v})$ one uses $\mathbf{y}' = \mathbf{y}\alpha^{\boldsymbol{\theta}(\mathbf{v})}$, $\alpha > 1$, given the vector of penalized violation values $\boldsymbol{\theta}(\mathbf{v})$. Below we compare both types of update and find that the multiplicative approach typically works better.

Weight smoothing: [Schuurmans and Southey, 2000] also introduce the idea of weight smoothing: after each update $\mathbf{y}' = \mathbf{y}\alpha^{\boldsymbol{\theta}(\mathbf{v})}$ the constraint weights are pulled back toward the population average $\overline{\mathbf{y}'} = \frac{1}{m}\sum_{i=1}^{m} y'_i$ using the rule $\mathbf{y}'' = \rho\mathbf{y}' + (1-\rho)\overline{\mathbf{y}'}$, $0 < \rho \leq 1$. This has the effect of increasing the weights of frequently satisfied constraints without requiring them to be explicitly violated (which led to a noticeable improvement in SDF's performance).

Exponentiated subgradient optimization (ESG): The final procedure we propose follows a standard subgradient optimization search with three main augmentations: *(1)* we use the augmented Lagrangian L_θ with a hinge penalty rather than a linear penalty, *(2)* we use multiplicative rather than additive updates, and *(3)* we use weight smoothing to uniformize weights without requiring constraints to be explicitly violated. The parameters of the final procedure are α, ρ, and a noise parameter η (see Figure 2).[3] Below we compare four variants of the basic method: ESG_h, ESG_ℓ (multiplicative updates with hinge and linear penalties respectively), and ASG_h, ASG_ℓ (additive updates with each penalty).[4]

The final ESG procedure differs from the SDF system of [Schuurmans and Southey, 2000] in many respects. The most important difference is that ESG can be applied to arbitrary BLP tasks, whereas SDF is applicable only to SAT problems. However, even if a generalized form of SDF could be devised, there would still remain minor differences: First, ESG uses a constant multiplier α for dual updates, whereas SDF uses an adaptive multiplier that obtains a minimum difference δ in $L(\mathbf{x}, \mathbf{y})$ for some primal search direction. Second, SDF uses a different penalty in its primal and dual phases (exponential versus hinge) whereas ESG uses the same hinge penalty in both phases. Third, ESG smooths all of the weights, whereas SDF only smooths weights on satisfied constraints. Finally, ESG includes a small probability of stepping out of local minima during its primal search, whereas SDF employs a deterministic primal search. Overall, these simplifications allow for a more streamlined implementation for ESG that yields a significant reduction in CPU overhead per step, while simultaneously allowing application to more general tasks.

$\text{ESG}_\theta(\alpha, \rho, \eta)$ procedure
Initialize $\mathbf{y} := \mathbf{1}$ and $\mathbf{x} :=$ random $\{-1, +1\}^n$
while solution not found **and** restart not reached
Primal search: Use greedy local search from $\mathbf{x}$ to find $\mathbf{x}'$ that locally minimizes $L_\theta(\mathbf{x}', \mathbf{y})$ (iff stuck, with probability η take a random move and continue); $\mathbf{v}' := C\mathbf{x}' - \mathbf{b}$.
Dual step: $\mathbf{y}' := \mathbf{y}\alpha^{\boldsymbol{\theta}(\mathbf{v}')}$; $\mathbf{y}'' := \rho\mathbf{y}' + (1-\rho)\overline{\mathbf{y}'}$.
Update: $\mathbf{x} := \mathbf{x}'$; $\mathbf{y} := \mathbf{y}''$

Figure 2: ESG procedure

4 SAT experiments

We consider applying the BLP techniques to SAT problems. An instance of SAT is specified by a set of clauses $\mathbf{c} = \{c_i\}_{i=1}^m$, where each clause c_i is a disjunction of k_i literals, and each literal denotes whether an assignment of "true" or "false" is required of a variable x_j. The goal is to find an assignment of truth values to variables such that every clause is satisfied on at least one literal. In our framework, an instance of SAT can be equivalently represented by the BLP

$$\min_{\mathbf{x}\in\{-1,+1\}^n} \mathbf{0}\cdot\mathbf{x} \quad \text{subject to} \quad C\mathbf{x} \leq \mathbf{k} - \mathbf{2} \tag{5}$$

where $\mathbf{0}$ and $\mathbf{2}$ are vectors of zeros and twos respectively, and

$$c_{ij} = \begin{cases} 1 & \text{if } x_j \text{ appears in a negative literal in } c_i \\ -1 & \text{if } x_j \text{ appears in a positive literal in } c_i \\ 0 & \text{otherwise} \end{cases}$$

An assignment of $+1$ to variable x_j denotes "true", and an assignment of -1 denotes "false". The idea behind this representation is that a clause c_i is encoded by a row vector $\mathbf{c}_i$ in C which has k_i nonzero entries corresponding to the variables occurring in c_i, along with their signs. A constraint is violated only when the $\{-1, +1\}$ assignment to $\mathbf{x}$ agrees with the coefficients in $\mathbf{c}_i$ on every nonzero entry, yielding a row sum of $\mathbf{c}_i \cdot \mathbf{x} = k_i$. If a single sign is flipped then the row sum drops by 2 and the constraint becomes satisfied.

Note that the zero vector means that the minimization component of this problem is trivialized. Nevertheless it remains a hard constraint satisfaction problem. The trivialized objective causes no undue difficulty to the methods introduced above, and therefore the BLP formulation allows us to accommodate both constraint satisfaction and constrained optimization problems in a common framework (albeit in a more restricted way than [Hooker *et al.*, 1999]).

For this problem we compared the subgradient optimization techniques ESG/ASG against state of the art local search methods: DLM, SDF, Novelty+, Novelty, WSAT, GSAT and HSAT. However, for reasons of space we report results only

[3] Software available at http://ai.uwaterloo.ca/~dale/software/esg/.

[4] Note that when using a linear penalty we employ an optimal primal search, but when using a hinge penalty we must resort to a greedy local search to approximately minimize $L(\mathbf{x}, \mathbf{y})$.

	Avg. Steps	Est. Opt. Steps	Fail %	Opt. sec
uf50	(100 problems)			
CPLEX (10 probs)	49	na	0	.186
$\text{ASG}_\ell(.12, 0, .02)$	500,000	na	100	na
$\text{ESG}_\ell(2.0, .05, .125)$	178,900	143,030	25	43.3
$\text{ASG}_h(.12, 0, .02)$	1,194	359	0.3	.027
$\text{ESG}_h(1.2, .99, .0005)$	215	187	0	.0009
uf100	(1000 problems)			
CPLEX (10 probs)	727	na	0	6.68
$\text{ASG}_h(.2, 0, .02)$	3,670	2,170	1.3	0.26
$\text{ESG}_h(1.15, .01, .002)$	952	839	0	.004
uf150	(100 problems)			
CPLEX (10 probs)	13,808	na	0	275.2
$\text{ASG}_h(.5, 0, .02)$	14,290	6,975	1.1	.071
$\text{ESG}_h(1.15, .01, .001)$	2,625	2,221	0	.011

Table 1: Comparison of general BLP methods on SAT

for DLM, SDF, and Novelty+, which represent the best of the last two generations of local search methods for SAT.[5] We ran these systems on a collection of (satisfiable) benchmark problems from the SATLIB repository.[6] Here the SAT solvers were run on their standard CNF input representations whereas the BLP solvers were run on BLP transformed versions of the SAT problems, as given above. (Although alternative encodings of SAT problems could affect the performance of BLP solvers [Beacham *et al.*, 2001] we find that the simple transformation described above yields reasonable results.)

To measure the performance of these systems we recorded wall clock time on a PIII 750MHz processor, average number of *primal* search steps ("flips") needed to reach a solution, and the proportion of problems not solved within 500K steps (5M on bw_large.c). To avoid the need to tune every method for a restart threshold we employed the strategy of [Schuurmans and Southey, 2000] and ran each method 100 times on every problem to record the distribution of solution times, and used this to estimate the minimum expected number of search steps required under an optimal restart scheme for the distribution [Parkes and Walser, 1996]. The remaining parameters of each method were manually tuned for every problem set. The parameters considered were: Novelty+(*walk, noise*), DLM(16 tunable parameters), $\text{SDF}(\delta, 1-\rho)$, $\text{ESG}_\theta(\alpha, 1-\rho, \eta)$ and $\text{ASG}_\theta(\alpha, 1-\rho, \eta)$.[7]

[5]Implementations of these methods are available at ai.uwaterloo.ca/∼dale, www.satlib.org, and manip.crhc.uiuc.edu.

[6]SATLIB is at www.satlib.org. We used three classes of problems in our experiments. The *uf* problems are randomly generated 3CNF formulae that are generated at the phase transition ratio of 4.3 clauses to variables. (These formulae have roughly a 50% chance of being satisfiable, but *uf* contains only verified satisfiable instances.) The *flat* problems are SAT encoded instances of hard random graph coloring problems. The remaining problems are derived from AI planning problems and the "all interval series" problem.

[7]For the large random collections (Tables 1 and 2: flat100–uf250) the methods were tuned by running on 10 arbitrary problems. Performance was then measured on the complete collection. Novelty+ parameters were tuned from the published values of [Hoos and Stützle, 2000]. DLM was tuned by running each of the 5 default parameter sets available at manip.crhc.uiuc.edu and choosing the best.

	Avg. Steps	Est. Opt. Steps	Fail %	Opt. sec
ais8	(1 problem)			
Novelty+(.4, .01)	183,585	20,900	9	.078
SDF(.0004, .005)	4,490	4,419	0	.083
DLM(pars1)	4,678	4,460	0	.044
$\text{ESG}_h(1.9, .001, .0006)$	4,956	4,039	0	.043
ais10	(1 problem)			
Novelty+(.3, .01)	434,469	340,700	79	1.64
SDF(.00013, .005)	15,000	13,941	0	.40
DLM(pars4)	18,420	14,306	0	.11
$\text{ESG}_h(1.9, .001, .0004)$	15,732	15,182	0	.23
ais12	(1 problem)			
Novelty+(.5, .01)	496,536	496,536	99	3.0
SDF(.0001, .005)	132,021	97,600	1	3.6
DLM(pars1)	165,904	165,904	3	2.5
$\text{ESG}_h(1.8, .001, .0005)$	115,836	85,252	10	1.7
bw_large.a	(1 problem)			
Novelty+(.4, .01)	9,945	8,366	0	.025
SDF(.0001, .005)	3,012	3,012	0	.057
DLM(pars4)	3,712	3,701	0	.028
$\text{ESG}_h(3, .005, .0015)$	2,594	2,556	0	.022
bw_large.b	(1 problem)			
Novelty+(.4, .01)	210,206	210,206	12	.82
SDF(.00005, .005)	33,442	33,255	0	1.30
DLM(pars4)	44,361	39,216	0	.34
$\text{ESG}_h(1.4, .01, .0005)$	33,750	26,159	0	.40
bw_large.c	(1 problem)			
Novelty+(.2, .01)	3,472,770	1,350,620	52	7
SDF(.00002, .005)	3,478,770	3,478,770	48	313
DLM(pars4)	3,129,450	2,282,900	32	41
$\text{ESG}_h(1.4, .01, .0005)$	1,386,050	875,104	5	39
logistics.c	(1 problem)			
Novelty+(.4, .01)	127,049	126,795	1	.38
SDF(.000009, .005)	15,939	15,939	0	1.40
DLM(pars4)	12,101	11,805	0	.13
$\text{ESG}_h(2.2, .01, .0025)$	8,962	8,920	0	.15
flat100	(100 problems)			
Novelty+(.6, .01)	17,266	12,914	.03	.019
SDF(.0002, .005)	7,207	6,478	0	.074
DLM(pars4)	9,571	8,314	0	.022
$\text{ESG}_h(1.1, .01, .0015)$	7,709	6,779	0	.022
flat200	(100 problems)			
Novelty+(.6, .01)	238,663	218,274	27	.34
SDF(.0001, .005)	142,277	115,440	7	1.56
DLM(pars4)	280,401	242,439	31	.77
$\text{ESG}_h(1.01, .01, .0025)$	175,721	134,367	14	.47
uf150	(100 problems)			
Novelty+(.6, .01)	6,042	3,970	0	.008
SDF(.00065, .005)	3,151	2,262	0	.023
DLM(pars4)	3,263	2,455	0	.008
$\text{ESG}_h(1.15, .01, .001)$	2,625	2,221	0	.011
uf200	(100 problems)			
Novelty+(.6, .01)	25,917	22,214	1.8	.048
SDF(.0003, .005)	14,484	11,304	.4	.134
DLM(pars4)	13,316	9,020	.1	.030
$\text{ESG}_h(1.23, .01, .003)$	10,583	7,936	.2	.039
uf250	(100 problems)			
Novelty+(.6, .01)	27,730	24,795	1.8	.055
SDF(.0002, .005)	18,404	13,626	.1	.177
DLM(pars4)	22,686	12,387	.2	.042
$\text{ESG}_h(1.15, .01, .003)$	13,486	10,648	.1	.054

Table 2: Comparison of ESG_h to specialized SAT methods

Table 1 compares the different BLP procedures on random problems from SATLIB. The most salient feature of these results is that the linear penalty performs poorly (as anticipated). Here, the traditional ASG_ℓ method was unable to solve any problems in the allotted steps. Table 1 also shows that multiplicative updates (ESG) were generally superior to additive updates (ASG) regardless of the penalty function used (at least on random SAT problems). Although details are omitted, we have also found that some smoothing ($1 - \rho \leq 0.01$) usually improves the performance of ESG but is generally less beneficial for ASG. CPLEX generally performs poorly in these experiments.

Table 2 shows the results of a larger scale comparison between ESG_h and state of the art local search methods for SAT. These results show that ESG_h tends to finds solutions in fewer primal search steps (flips) than other procedures. However, ESG_h's time per flip is about 1.5 to 2 times higher than DLM and about 2 to 4 times higher than Novelty+, which means that its overall run time is only comparable to these methods. Nevertheless, ESG_h, DLM and SDF obtain a large reduction in search steps over Novelty+ on the structured ais and planning problems (with the notable exception of bw_large.c). As a result, DLM and ESG_h both tend to demonstrate better run times than Novelty+ on these problems. All three methods (Novelty+, DLM and ESG_h) demonstrate similar run times on the random uf and flat problems.

Note that none of these SAT procedures (other than ESG/ASG and CPLEX) can be applied to general BLP problems without modification.

5 CA experiments

The second problem we considered is optimal winner determination in combinatorial auctions (CA). This problem introduces a nontrivial optimization objective that is not present in SAT. However, the subgradient optimization approach remains applicable, and we can apply the same ESG/ASG methods to this task without modifying them in any way. (Nevertheless, we did conduct some implementation specializations to gain improvements in CPU times on SAT problems.) The CA problem has been much less studied in the AI literature, but interest in the problem is growing rapidly.

An instance of the optimal winner determination problem in combinatorial auctions (CA) is given by a set of items $\mathbf{c} = \{c_i\}_{i=1}^m$ with available quantities $\mathbf{q} = \{q_i\}_{i=1}^m$, and a set of bids $\mathbf{z} = \{z_j\}_{j=1}^n$ which offer amounts $\mathbf{v} = \{v_j\}_{j=1}^n$ for a specified subset of items. We can represent the bid requests in a constraint matrix C where

$$c_{ij} = \begin{cases} 1 & \text{if bid } z_j \text{ requests item } c_i \\ 0 & \text{otherwise} \end{cases}$$

The problem then is to find a set of bids that maximizes the total revenue subject to the constraint that none of the item quantities are exceeded. If $z_j \in \{0, 1\}$ this problem can be expressed as the BLP

$$\max_{\mathbf{z} \in \{0,1\}} \mathbf{v} \cdot \mathbf{z} \quad \text{subject to} \quad C\mathbf{z} \leq \mathbf{q} \tag{6}$$

However it is not in our canonical $\{-1, +1\}$ form. To transform it to the canonical form we use the substitutions $\mathbf{x} = 2\mathbf{z} - 1$, $\mathbf{a} = -\mathbf{v}/2$ and $\mathbf{b} = 2\mathbf{q} - C\mathbf{1}$. The minimum solution to the transformed version of this problem can then be converted back to a maximum solution of the original CA.

For this task we compared the ESG/ASG algorithms to CPLEX and Casanova [Hoos and Boutilier, 2000], a local search method loosely based on Novelty+.[8] The problems we tested on were generated by the CATS suite of CA problem generators [Leyton-Brown *et al.*, 2000], which are intended to model realistic problems. However, our results indicate that these tend to be systematically easy problems for all the methods we tested, so we also tested on earlier artificial problem generators from the literature [Sandholm, 1999; Fujishima *et al.*, 1999]. Some of these earlier generators were shown to be vulnerable to trivial algorithms [Andersson *et al.*, 2000] but some still appear to generate hard problems. However, to push the envelope of difficulty further, we also encoded several hard SAT problems as combinatorial auctions by using an efficient polynomial (quadratic) reduction from SAT to CA. Unfortunately, space limitations preclude a detailed description of this reduction, but our results show that these converted SAT problems are by far the most difficult available at a given problem size.

To measure the performance of the various methods, we first solved all of the problems using CPLEX and then ran the local search methods until they found an optimal solution or timed out. Although we acknowledge that this method of reporting ignores the anytime performance of the various methods, it seems sufficient for our purposes. To give some indication of anytime behavior, we recorded the fraction of optimality achieved by the local search methods in cases where they failed to solve the problem within the allotted time.

Table 3 shows a comparison of the different subgradient optimization procedures on one of the CATS problem classes. These results show that the linear penalty is once again weaker than the hinge (but to a lesser extent than on SAT problems). Interestingly, additive updates appear to work as well as multiplicative updates on these problems (although the performance of ASG begins to weaken on harder constraint systems such as those encountered in Table 5).

Table 4 shows the results of a larger comparison between ESG/ASG, Casanova and CPLEX on the CATS problems and an artificial problem. These results show that there is very little reason not to use CPLEX to solve these problems in practice.[9] CPLEX, of course, also proves that its answers are optimal, unlike local search. Nevertheless, the results show that ESG_h and ASG_h are roughly competitive with Casanova, which is a specialized local search technique for this task.

[8] Casanova is specialized to single unit CA problems ($\mathbf{q} = \mathbf{1}$), so we restrict attention to this class. However, ESG/ASG and CPLEX can be applied to multi-unit CA problems without modification, and hence are more general. CPLEX was run with the parameter settings reported in [Andersson *et al.*, 2000]. Casanova uses the same parameters as Novelty+, *walk* and *noise*.

[9] The CPLEX timings show time to first discover the optimum, not prove that it is indeed optimal. Also note that even though "steps" are recorded for each of the methods, they are incomparable numbers. Each method uses very different types of steps (unlike the previous SAT comparison in terms of "flips") and are reported only to show method-specific difficulty.

Note that the large score proportions obtained by all methods when they fail to find the optimal solution follows from the fact that even simple hill-climbing strategies tend to find near optimal solutions in a single pass [Holte, 2001]. It appears that most of the difficulty in solving these problems is in gaining the last percentage point of optimality.

Finally, Table 5 shows that Casanova demonstrates inferior performance on the more difficult SAT→CA encoded problems. Nevertheless all methods appear to be challenged by these problems. Interestingly, CPLEX slows down by a factor of 400 when transforming from the direct SAT encoding of Section 4 to the SAT→CA encoding used here. By comparison the ESG method slows down by a factor of 20K.

6 Conclusion

Although we do not claim that the methods we have introduced exhibit the absolute best performance on SAT or CA problems, they nevertheless compete very well with the state of the art techniques on both types of problem. Therefore, we feel that the simple Lagrangian approach introduced here might offer a useful starting point for future investigation beyond the myriad WalkSAT variants currently being pursued in AI. Among several directions for future work are to investigate other general search ideas from the OR literature, such as tabu search [Glover, 1989], and compare to other local search methods for ILP problems [Resende and Feo, 1996; Voudouris and Tsang, 1996] (which do not appear to be competitive on SAT problems, but still offer interesting perspectives on ILP problems in general).

A Weak duality

It is insightful to see why $D(\mathbf{y}) \leq P^*$ for all $\mathbf{y} \geq \mathbf{0}$. In fact, it is almost immediately obvious: Note that for any $\mathbf{y} \geq \mathbf{0}$ we have

$$\begin{aligned} D(\mathbf{y}) &= \min_{\mathbf{x}\in\{-1,+1\}^n} L(\mathbf{x},\mathbf{y}) \\ &\leq \mathbf{a}\cdot\mathbf{x} + \mathbf{y}^\top(C\mathbf{x}-\mathbf{b}) \quad \text{for all } \mathbf{x}\in\{-1,+1\}^n \\ &\leq \mathbf{a}\cdot\mathbf{x} \quad \text{for all } \mathbf{x} \text{ such that } C\mathbf{x}\leq\mathbf{b} \quad (\text{since } \mathbf{y}\geq\mathbf{0}) \end{aligned}$$

and hence $D(\mathbf{y}) \leq P^*$. From this one can see that the constraint $\mathbf{y} \geq \mathbf{0}$ is imposed precisely to ensure a lower bound on P^*.

B Subgradient direction

It is also easy to see that the vector of constraint violation values $\mathbf{v_y} = C\mathbf{x_y} - \mathbf{b}$ (where $\mathbf{x_y} = \arg\min_{\mathbf{x}\in\{-1,+1\}^n} L(\mathbf{x},\mathbf{y})$) must yield a subgradient of D at $\mathbf{y}$:

$$\begin{aligned} D(\mathbf{z}) &= \min_{\mathbf{x}\in\{-1,+1\}^n} \mathbf{a}\cdot\mathbf{x} + \mathbf{z}^\top(C\mathbf{x}-\mathbf{b}) \\ &\leq \mathbf{a}\cdot\mathbf{x_y} + \mathbf{z}^\top(C\mathbf{x_y}-\mathbf{b}) \ (\text{since } L(\mathbf{x_z},\mathbf{z}) \leq L(\mathbf{x_y},\mathbf{z})) \\ &= \mathbf{a}\cdot\mathbf{x_y} + \mathbf{y}^\top(C\mathbf{x_y}-\mathbf{b}) + (\mathbf{z}-\mathbf{y})^\top(C\mathbf{x_y}-\mathbf{b}) \\ &= D(\mathbf{y}) + (\mathbf{z}-\mathbf{y})^\top\mathbf{v_y} \end{aligned}$$

Acknowledgments

Research supported by NSERC and CITO. Thanks to Holger Hoos and Craig Boutilier for helpful comments and providing access to Casanova. Thanks also to Peter van Beek and the anonymous referees for many helpful suggestions. The assistance of István Hernádvölgyi is also warmly appreciated.

	Avg. sec	Avg. Steps	Fail %	% Opt.
CATS-regions	(100 problems)			
$\text{ESG}_h(1.9,.1,.01)$	7.2	1416	4.1	99.93
$\text{ASG}_h(.045,0,.01)$	12.7	2457	7.9	99.86
$\text{ESG}_\ell(1.3,.9,.01)$	64.7	7948	77	88.11
$\text{ASG}_\ell(.01,0,.01)$	48.1	9305	90	93.96

Table 3: Comparison of subgradient optimization methods

	Avg. sec	Avg. Steps	Fail %	% Opt.
CATS-regions	(100 problems)			
CPLEX	6.7	64117	0	100
Casanova$(.5,.17)$	4.2	1404	3.4	99.95
$\text{ESG}_h(1.9,.1,.01)$	7.2	1416	4.1	99.93
$\text{ASG}_h(.045,0,.01)$	12.7	2457	7.9	99.86
CATS-arbitrary	(100 problems)			
CPLEX	22	9510	0	100
Casanova$(.04,.12)$	9	2902	0.47	99.98
$\text{ESG}_h(2.1,.05,.5)$	33	7506	4.21	99.95
$\text{ASG}_h(.04,0,.01)$	30	6492	4.87	99.87
CATS-matching	(100 problems)			
CPLEX	1.30	499	0	100
Casanova$(.3,.17)$	.17	109	0	100
$\text{ESG}_h(1.7,.05,0)$	.16	215	0	100
$\text{ASG}_h(.9,0,.8)$	.73	1248	0	100
CATS-paths	(100 problems)			
CPLEX	25	1	0	100
Casanova$(.5,.15)$	26	49	0	100
$\text{ESG}_h(1.3,.05,.4)$	28	2679	2.5	99.99
$\text{ASG}_h(.01,0,.15)$	75	5501	6.8	99.96
CATS-scheduling	(100 problems)			
CPLEX	15	1426	0	100
Casanova$(.5,.04)$	44	7017	19.9	99.87
$\text{ESG}_h(1.35,.05,.015)$	65	11737	41	99.68
$\text{ASG}_h(.015,0,.125)$	58	12925	44.2	99.51
Decay-200-200-.75	(100 problems)			
CPLEX	1.1	2014	0	100
Casanova$(.5,.17)$	0.5	2899	2	99.97
$\text{ESG}_h(14.5,.045,.45)$	1.8	24466	96.3	96.91
$\text{ASG}_h(.04,0,.5)$	1.7	24939	99.6	91.49

Table 4: Results on CATS and synthetic problems

	Avg. sec	Avg. Steps	Fail %	% Opt.
SAT(uf50)→CA	(10 problems)			
CPLEX	42	754	0	100
Casanova$(.5,.11)$	468	9.6×10^6	90	99.36
$\text{ESG}_h(30,.05,.1)$	31	1.7×10^6	10	99.95
$\text{ASG}_h(5,0,.5)$	173	8.6×10^6	80	99.40
SAT(uf75)→CA	(10 problems)			
CPLEX	666	5614	0	100
Casanova$(.5,.11)$	800	1.0×10^7	100	98.46
$\text{ESG}_h(41,.05,.1)$	165	6.2×10^6	60	99.81
$\text{ASG}_h(10,0,.5)$	291	1.0×10^7	100	99.17

Table 5: Results on hard SAT→CA encoded problems

References

[Andersson *et al.*, 2000] A. Andersson, M. Tenhunen, and F. Ygge. Integer programming for combinatorial auction winner determination. In *Proceedings ICMAS-00*, 2000.

[Beacham *et al.*, 2001] A. Beacham, X. Chen, J. Sillito, and P. van Beek. Constraint programming lessons learned from crossword puzzles. In *Proc. Canadian AI Conf.*, 2001.

[Bertsekas, 1995] D. Bertsekas. *Nonlinear Optimization*. Athena Scientific, 1995.

[Davenport *et al.*, 1994] A. Davenport, E. Tsang, C. Wang, and K. Zhu. GENET: A connectionist architecture for solving constraint satisfaction problems. In *Proceedings AAAI-94*, pages 325–330, 1994.

[Everett, 1963] H. Everett. Generalized Lagrange multiplier method for solving problems of the optimal allocation of resources. *Operations Res.*, 11:399–417, 1963.

[Fisher, 1981] M. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Sci.*, 27:1–18, 1981.

[Frank, 1997] J. Frank. Learning short-tem weights for GSAT. In *Proceedings IJCAI-97*, pages 384–391, 1997.

[Fujishima *et al.*, 1999] Y. Fujishima, K. Leyton-Brown, and Y. Shoham. Taming the computational complexity of combinatorial auctions: optimal and approximate approaches. In *Proceedings IJCAI-99*, pages 548–553, 1999.

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

[Gomes, 2000] C. Gomes. Structure, duality, and randomization: Common themes in AI and OR. In *Proceedings AAAI-00*, pages 1152–1158, 2000.

[Held and Karp, 1970] M. Held and R. Karp. The travelling salesman problem and minimum spanning trees. *Operations Res.*, 18:1138–1162, 1970.

[Holte, 2001] R. Holte. Combinatorial auctions, knapsack problems, and hill-climbing search. In *Proceedings Canadian AI Conference*, 2001.

[Hooker *et al.*, 1999] J. Hooker, G. Ottosson, E. Thorsteinsson, and H.-K. Kim. On integrating constraint propagation and linear programming for combinatorial optimization. *Proceedings AAAI-99*, pages 136–141, 1999.

[Hoos and Boutilier, 2000] H. Hoos and C. Boutilier. Solving combinatorial auctions using stochastic local search. In *Proceedings AAAI-00*, pages 22–29, 2000.

[Hoos and Stützle, 2000] H. Hoos and T. Stützle. Local search algorithms for SAT: An empirical evaluation. *J. Automat. Reas.*, 24:421–481, 2000.

[Hoos, 1999] H. Hoos. On the run-time behavior of stochastic local search algorithms for SAT. In *Proceedings AAAI-99*, pages 661–666, 1999.

[Kautz and Selman, 1996] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings AAAI-96*, pages 1194–1201, 1996.

[Kautz and Walser, 1999] H. Kautz and J. Walser. State-space planning by integer optimization. *Proceedings AAAI-99*, pages 526–533, 1999.

[Kivinen and Warmuth, 1997] J. Kivinen and M. Warmuth. Exponentiated gradient versus gradient descent for linear predictors. *Infor. Comput.*, 132:1–63, 1997.

[Larsson *et al.*, 1996] T. Larsson, M. Patriksson, and A.-B. Stromberg. Conditional subgradient optimization—theory and applications. *Euro. J. Oper. Res.*, 88:382–403, 1996.

[Lau *et al.*, 2000] H. Lau, A. Lim, and Q. Liu. Solving a supply chain optimization problem collaboratively. *Proceedings AAAI-00*, pages 780–785, 2000.

[Leyton-Brown *et al.*, 2000] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proc. EC-00*, 2000.

[Magazine and Oguz, 1984] M. Magazine and O. Oguz. A heuristic algorithm for the multidimensional knapsack problem. *Euro. J. Oper. Res.*, 16:319–326, 1984.

[Martin, 1999] R. Martin. *Large Scale Linear and Integer Optimization*. Kluwer, 1999.

[McAllester *et al.*, 1997] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings AAAI-97*, pages 321–326, 1997.

[Parkes and Walser, 1996] A. Parkes and J. Walser. Tuning local search for satisfiability testing. In *Proceedings AAAI-96*, pages 356–362, 1996.

[Resende and Feo, 1996] M. Resende and T. Feo. A GRASP for satisfiability. In *Cliques, Coloring, and Satisfiability*, DIMACS series v.26, pages 499–520. AMS, 1996.

[Sandholm, 1999] T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *Proceedings IJCAI-99*, pages 542–547, 1999.

[Schuurmans and Southey, 2000] D. Schuurmans and F. Southey. Local search characteristics of incomplete SAT procedures. In *Proc. AAAI-00*, pages 297–302, 2000.

[Selman *et al.*, 1994] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings AAAI-94*, pages 337–343, 1994.

[Thornton and Sattar, 1999] J. Thornton and A. Sattar. On the behavior and application of constraint weighting. In *Proceedings CP-99*, pages 446–460, 1999.

[Vossen *et al.*, 1999] T. Vossen, M. Ball, A. Lotem, and D. Nau. On the use of integer programming models in AI planning. *Proceedings IJCAI-99*, pages 304–309, 1999.

[Voudouris and Tsang, 1996] C. Voudouris and E. Tsang. Partial constraint satisfaction problems and guided local search. In *Proceedings PACT-96*, pages 337–356, 1996.

[Walser, 1999] J. Walser. *Integer Optimization by Local Search*. Springer-Verlag, 1999.

[Wu and Wah, 2000] Z. Wu and W. Wah. An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In *Proceedings AAAI-00*, pages 310–315, 2000.

A New Method For The Three Dimensional Container Packing Problem

Andrew Lim and Wang Ying
Department of Computer Science, National University of Singapore
3 Science Drive 2, Singapore 117543

Abstract

A new algorithm for solving the three dimensional container packing problem is proposed in this paper. This new algorithm deviates from the traditional approach of wall building and layering. It uses the concept of "building growing" from multiple sides of the container. We tested our method using all 760 test cases from the OR-Library. Experimental results indicate that the new algorithm is able to achieve an average packing utilization of more than 87%. This is better than the results reported in the literature.

1 Introduction

Packing boxes into containers is a common problem faced by the logistics and transportation industries. The three dimensional (3D) packing problem is a difficult extension of the two dimensional (2D) stock cutting problem and the one dimensional (1D) bin packing problem. Even the 1D bin packing problem is known to be NP-Complete. For a survey of packing problems, please refer to [K.A. and W.B., 1992]. While pseudo-polynomial time algorithms have been derived for the 1D packing problem, no such results have been derived for the 3D packing problem.

The 3D packing problem can be divided into two main variants. The first variant is: given a list of boxes of various sizes, pack a subset of these boxes into one container to maximize the volume utilization of the container. The second variant is: given a list of boxes of various sizes, pack all the boxes into as few containers as possible. In addition, constraints, such as orientation restriction, weight distribution, locality restriction, may be imposed on the boxes.

In this paper, we study the packing problem that maximizes the volume utility of a single container with orientation restrictions on the boxes.

Many different heuristics have been developed to "optimize" volume utilization in a single container. [J.A and B.F., 1980] proposed a primitive "wall-building" method. [Gehring H. and M., 1990] proposed an improved "wall-building" heuristic. [T.H. and Nee, 1992] proposed a layering approach. [Han C.P. and J.P., 1989] provided an algorithm for packing identical boxes, which is a combination of "wall-building" and layering. [B.K. and K., 1993] proposed a data structure for representing the boxes and free spaces, which is used in their heuristic. [E.E. and M.D., 1990] presented a comparative evaluation of different heuristic rules for freight container loading with the objective of minimizing the container length needed to accommodate a given cargo. [W.B., 1991] examined a number of heuristic rules for the problem. The emphasis on both wall building and layering is to create walls or layers of boxes as flat as possible. At all times, the methods attempt to retain a flat forward packing face. The existing methods most mirror the hand-packing process, which is a natural way to solve the problem. However, they put many restrictions on the location of the boxes which may not be feasible to the actual application. Another problem of the existing approaches is that they may leave a thin layer of non-useful empty spaces near the entrance or the ceiling of the container, which results in space underutilization. Also, the papers in the existing literature used data that are not easily accessible, hence, it is relatively hard to compare the various methods quantitatively.

In this paper, we propose a new method. Boxes are packed into container in "building-growing" fashion instead of "wall building" manner. Furthermore, each wall of the container can be treated as base to locate the boxes, which provides more flexibility. We tested our program on all the 760 test cases in [OR-Lib,]. Our experimental results indicate that we are able to achieve an average packing utilization more than 87%.

2 Problem Specification

We are given a set of rectangular boxes $B = \{b_1, \ldots, b_n\}$ and a rectangular container C. Let x_i, y_i and z_i represent the three dimensions of the box b_i, and X, Y, and Z represent the three dimensions of the container.

The objective of the problem is to select a subset $S \subseteq B$ and assign a position to each box, $b_j \in S$, in the 3D space of container C such that $\sum_{b_j \in S} x_j \times y_j \times z_j$ is maximized subjected to the constraints that all boxes must be totally contained in C, no two boxes intersect in 3D space and every edge of the packed box must be parallel or orthogonal to the container walls.

3 Multi-Directional Building-Growing Algo

The idea of this 3D packing algorithm [Wang, 2001] comes from the process of constructing a building. When erecting a building, we construct the basement on the ground first, then the building will "rise up" floor by floor. Here, the box will play the role of the floor, and the wall of the container will act as the ground. Boxes are placed on the wall of the container first as it builds the basement on the ground. After which, other boxes will be placed on top of the basement boxes. Following this process, boxes will be placed one on top of another, just like the construction of a new building. Every wall of the container can be treated as the ground for boxes to stack on. However, the users reserve the right to choose the walls to act as the ground. Note that, this algorithm does not take into consideration whether packed boxes form a flat layer or not, which is the main idea of the "wall building" approach. Figure 1 illustrates the main idea of this algorithm.

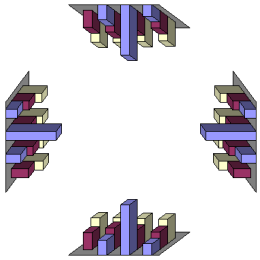

Figure 1: Multi-Directional Building-Growing Approach

3.1 Formulation of the Algorithm

Overall Packing Process

After getting the option from the user for the acting-as-ground walls, this algorithm reads in the container and box information, including length, width, height, orientation and quality information, from the data file. The algorithm then orders the boxes according to certain criteria. Subsequently, actual packing will be performed based on the best matching between base area of boxes and empty spaces. The following pseudo code presents the overall algorithm.

main{
  1. **get_wall_option()**
  2. *read in the container information (length,width,height)*
  3. *for(every box of type b)*{
    1.*read in b's information(length,width,height,orientation,quantity)*
    2.*bn*=**create_box_nodes(box b)**
    3.**insert_box_node(box_node bn)**
  }
  4. **pack()**
}

get_wall_option()

Users are free to choose which of the walls of a container to be used as ground/base. Basically, all the empty spaces created from one ground(wall) form an empty space list. There are at most six empty space lists as shown in Figure 2.

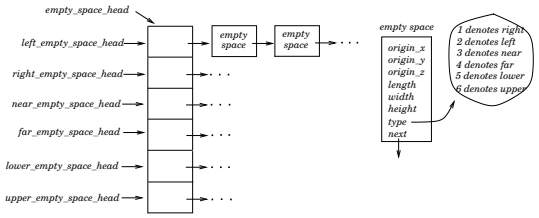

Figure 2: Empty Space List and Empty Space Object Data Structure

An empty space in each empty space list has its own origin and coordinate system as shown in Figure 3. Here the lower coordinate system is the "normal" coordinate system.

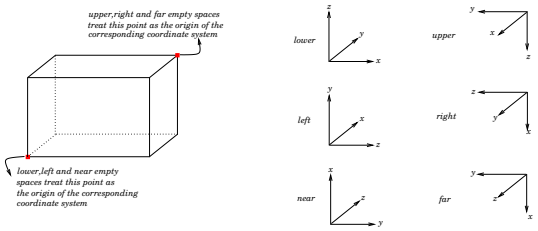

Figure 3: Empty Space Coordinate System Representation

The initial value of each empty space list head is the whole container. If the user does not choose wall i as the ground, then empty space list i's head should be pointing to null. Note that each empty space list i only takes care of all the empty spaces using wall i as base, without knowing the existence of the other empty space lists. Because of the exact and accurate representation, empty spaces may overlap with each other.

create_box_nodes(box b)

Every box has three attributes namely, length, width and height. Length is measured along the x-axis, so is width along the y-axis and height along the z-axis. And the area surrounded by length and width is called the base area of a box. There is a unique *id* assigned to each box, yet, there are six orientations associated with it (see Figure 4. Note that a number is used to denote each edge of the box.).

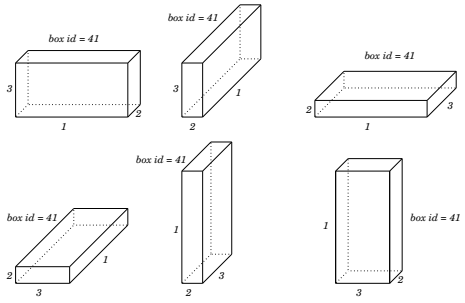

Figure 4: Six Orientations Associated with One Box

For each type of box (same type of box means boxes with same length, width and height), at most six box nodes will

	box1	box2	box3	box4	box5	box6
id	41	41	41	41	41	41
length	1	2	1	3	2	3
width	2	1	3	1	3	2
height	3	3	2	2	1	1

Table 1: *Id* and Attributes Associated with Box in Figure 4

be created for them but with the same *id*. Each box node represents a possible orientation for one type of box. If a certain orientation is not allowed, this algorithm will simply omit the box node of that orientation.

insert_box_node(box_node bn)
Whenever a box information is obtained from the data file, a new box node will be created and inserted to the box_node_list based on the function below. Given a newly created box node bn_2, this algorithm goes through the box_node_list, and for each existing box node bn_1, calculates the priority number for bn_1 and bn_2 respectively as follows:

$$priority(bn_1) = vol\_per \times \frac{volumn(bn_1)}{volumn(bn_2)} + dim\_per \times \frac{big(bn_1)}{big(bn_2)}$$

$$priority(bn_2) = vol\_per \times \frac{volumn(bn_2)}{volumn(bn_1)} + dim\_per \times \frac{big(bn_2)}{big(bn_1)}$$

where parameters vol_per and dim_per denote the volume priority percentage and the biggest dimension priority percentage respectively; routines $volume(bn)$ and $big(bn)$ return the volume and the biggest dimension of box b inside box node bn respectively. After getting $priority(bn_1)$ and $priority(bn_2)$, the algorithm compares these two numbers:

- if $priority(bn_2) > priority(bn_1)$, insert bn_2 to the front of bn_1.
- otherwise, continue to compare $priority(bn_2)$ with the priority number of the box node behind bn_1.

According to the above, a box node nearer to the head possesses either larger volume or one bigger dimension, hence, has higher priority to be packed into the container first. Note that boxes with the same *id* but with different orientations are located adjacent to each other, and they have the same priority to be packed into the container, because they are actually the same box.

pack()
In every attempt to pack one box into the container, the algorithm finds a most suitable empty space for each of the first i box nodes with the sane priority number,in terms of the percentage of the base area of that box occupies the base area of the empty space. Inside all these i pairs of box node and empty space, the algorithm finds one pair (bn, oe) with the greatest base area occupation percentage p,

- remove bn, if $p > 1$.
- pack bn, if $packing\_per \leq p \leq 1$.
- decide to pack or remove bn carefully, if $0 < p < packing\_per$

The following pseudo code illustrates the packing process of the algorithm.

while(box_node_head!=null){
  for(box node bn_i with same priority as box_node_head){
    for(every empty space oe in the 6 empty space lists){
      percentage(bn_i)=$\frac{base\_area(bn_i)}{base\_area(oe)}$
    }
  }
  find one pair (bn,oe) with greatest p=percentage(bn)
  if($p > 1$) {
    //bn cannot be packed into any empty space
    remove bn
    //because bn is too big to be packed into any empty space now
    //and empty space will become smaller and smaller, there will
    //be no chance for bn to be packed into any empty space later
  }
  if($packing\_per \leq p \leq 1$){
    1.create a packed box pb according to oe's coordinate system, then convert it to the normal coordinate system and insert it into packed_box_list
    2.delete one box id from box node bn
    *3.***update_empty_space(empty_space oe,packed_box pb)**
  }
  if(0< p <packing_per){
    //in this case, we cannot find a good fit between bn and oe,
    //so we need to decide whether we can drop the first i box
    //nodes with the same priority number
    if(after removing the first i box nodes and $\frac{\sum volume(the\ rest\ unpacked\ boxes\ +\ the\ packed\ boxes)}{volume(container)} \geq$ *moving_per){*
      remove the first i box nodes
      //because without these few difficult-to-be-packed box nodes,
      //we may still get good utilization
    }
    else{
      1.create a packed box pb according to oe's coordinate system then convert it to the normal cooridnate system and insert it into packed_box_list
      2.delete one box id from box node bn
      *3.***update_empty_space(empty_space oe,packed_box pb)**
    }
  }
}

update_empty_space(empty_space oe,packed_box pb)
After a box is packed into the container, the original set of empty spaces will be modified by the newly packed box pb. Moreover, pb will not only affect the empty space *oe* it resides in, but also the empty spaces nearby *oe*, because, in order to represent the empty spaces effectively, the empty spaces overlap. The responsibility of this routine is to update the empty spaces after one box is placed into the container. This process can be performed using the following two steps:

Step 1: Change pb to the coordinate system which *oe* lies in and absorb the new empty spaces out of *oe* (Figure 5).

Updating *oe*:

1. Remove *oe* from its empty space list
2. If (es_1, es_2, es_3 are not inside the empty_space list *oe* resides in)

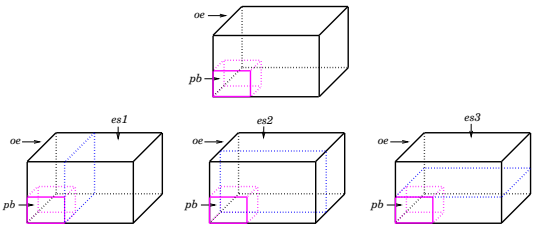

Figure 5: Update Empty Space oe which Provides Empty Space for Packed Box pb

insert it(them) into the empty_space list *oe* resides in

Step 2: For every empty space e in the six empty space lists,

1. Convert pb to the coordinate system which e is in
2. Test whether there is overlap between e and pb, and update e accordingly

All the possible overlap cases between e and pb will be disscussed below and the update method will be given by the end.
case 1: pb intersects one corner of e.
e.g. pb intersects the lower left near corner of e.
case 2: pb intersects one entire edge of e.
e.g. pb intersects the left edge of upper plane of e.
case 3: pb intersects the inner part of one edge of e.
e.g. pb intersects part of lower left edge of e.
case 4: pb gets into e from one plane of it without touching any corner point of e.
e.g. pb gets into e from the left plane of e.
case 5: pb cuts away one entire plane of e.
e.g. pb cuts away the whole left plane of e.
case 6: pb crosses one pair of parallel edges of e without touching any corner point of e
e.g. pb crosses the upper horizontal pair of edges of e.
case 7: pb passes through one pair of parallel planes of e without touching any corner point of e
e.g. pb passes through the near-far planes of e.
case 8: pb cuts e into two new empty_space.
e.g. pb cuts e into two new empty_space from vertical direction.
case 9: the whole pb resides inside e.
case 10: pb covers the whole e.

Updating e in each case above:

1. Remove e from its empty space list
2. If (newly created empty spaces es_i are not inside the empty_space list e resides in)
   insert it(them) into the empty_space list e resides in

3.2 Experimental Results

This algorithm was successfully implemented using the Java Programming Language. All the experiments were run on an Alpha 600MHz machine running the Digital Unix operating system. The OR-Library [OR-Lib,] has a collection of test data for a variety of OR problems. There are nine sets of test data with 760 test cases for the 3D packing problem. All the test data in [OR-Lib,] have been tested.

The algorithm was tuned using various parameter settings. The packing result for test file 8 is omitted in all result presentations, because of the fact that we can pack every box into the container in most cases and the size of the container is much larger than the total size of the boxes to be packed into the container.

The best packing result in terms of container volume utilization is shown in Figure 6. The parameter and wall combination used for the best packing result is shown in Table 2.

Parameter	Wall Combination
packing_vol=0.95	left wall
moving_per=1.0	right wall
vol_per=0.9	far wall
dim_per=0.1	lower wall

Table 2: Best Parameter Setting and Wall Combination

In the experimental results shown in Figure 6, the X-axis represents the test case index and the Y-axis represents the container volume utilization.

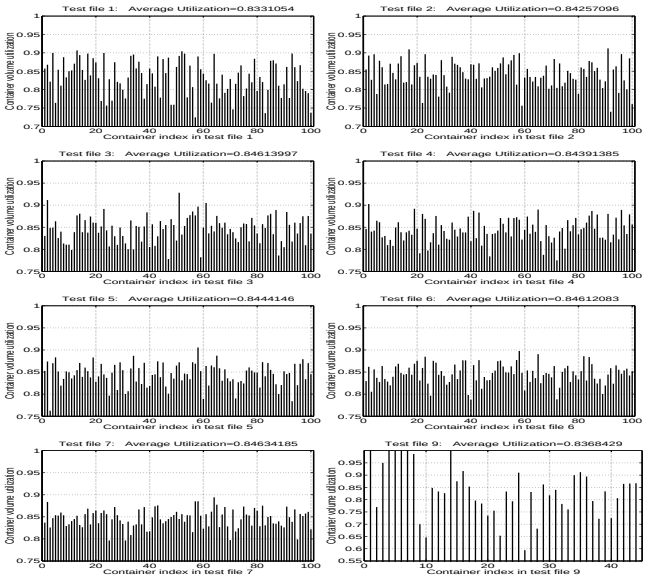

Figure 6: Best Experimental Results for Test File 1-7,9 Using Multi-Directional Building-Growing Approach

Using this parameter setting and wall combination, the average container volume utilization for all the available test cases in the OR-Library (except dest file 8) can reach as high as 0.84285194. This utilization is comparable to the published result of the existing literature. Moreover, the special advantage of this algorithm is its speed as it needs about two seconds to pack one container.

On the other hand, the container volume utilization can still be improved for the price of efficiency. In the next section, an improvement phase is introduced.

4 Incorporating a Look-Ahead Mechanism

In the packing process of Multi-Directional Building-Growing (MDBG) approach, the algorithm only chooses one pair of best fit box and empty space in terms of base area occupation percentage among the most difficult-to-be-packed boxes. MDBG Algorithm only takes care of the local information, and the decision made by this algorithm doesn't contain any knowledge for the future. The look-ahead mechanism not only considers the current pairing, but also looks at its potential effect.

4.1 Look-Ahead Improvement Strategy

The strategy will choose k pairs of boxes and empty spaces with the greatest base area occupation percentage. Each one of these k pairs of boxes and empty spaces will be assumed to be the chosen pair for this attempt and be temporarily packed into the container, and the packing process is continued using the MDBG approach from this attempt onwards until all the boxes have been packed into the container or no more boxes can be packed into it. Then based on the final container volume utilization, the improvement strategy will choose one pair of the box and empty space from the initial k pairs, which gives best final container volume utilization. This chosen pair of box and empty space will then be used as the actual packing for this attempt. This pair of box and empty space satisfies both best fit currently and best potential effect in the future. Figure 7 illustrates the look-ahead idea.

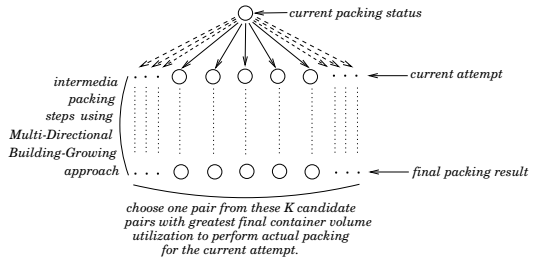

Figure 7: Illustration of the Look-Ahead Improvement Phase

The following is the pseudo-code for the look-ahead improvement strategy.

```
newpack(){
  while(there's box in the box node list){
    //attempt to pack one box into the container
    //1.get K boxes with greatest base area occupation percentage
    for(every box b in the box node list){
      for(every empty space in the 6 empty space lists){
        find a best fit empty space e for b with greatest
        base area occupation percentage p
      }
      if(p > 1){
        //b cannot be packed into any empty space
        remove b
      }
      else if(top K base area occupation percentages<p≤1){
        replace e and b into the top K best fit pairs
      }
    }
    //2. using MDBG approach pack the K candidate pairs
    for(i=1..k){
      1.temporarily pack b_i to empty space e_i at this attempt
      2.continue packing using pack() in section 3
      3.record down the final container volume utilization at u_i
    }
    //3.get one box with the greatest final utilization from the K
    //  candidates, and use this box in the actual packing of this
    //  attempt
    find one box b from b_i and one empty space e from e_i with
    greatest final container volume utilization u
    pack b into e permanently
  }
}
```

4.2 Experimental Results

In the last section, using MDBG approach, we have experimented with all the test data using all possible wall combinations for many possible parameter settings. The best packing result comes from the wall combination: left wall, right wall, far wall, lower wall; and the parameter setting: $packing\_vol = 0.95, moving\_per = 1.0, vol\_per = 0.9, dim\_per = 0.1$. Based on this experience, the Look-Ahead mechanism is tested on all data files using this parameter setting and wall combination, and let k go from 1 to 6. The results are summarized in the tables below:

	k=1	k=2	k=3	k=4	k=5	k=6
file1	0.81054	0.83601	0.85984	0.86713	0.87859	0.88356
file2	0.81704	0.83970	0.86335	0.86457	0.87433	0.87576
file3	0.82474	0.83616	0.85437	0.86468	0.86952	0.87036
file4	0.83225	0.84148	0.85376	0.85925	0.87081	0.87068
file5	0.83525	0.84282	0.85540	0.86260	0.86822	0.87072
file6	0.83403	0.84009	0.85665	0.86275	0.86534	0.86679
file7	0.83761	0.84349	0.85048	0.86072	0.86260	0.86575
file9	0.84885	0.87370	0.88711	0.89616	0.90393	0.90656
average	0.82862	0.84196	0.85809	0.86505	0.87193	0.87399

Table 3: Packing Result With Look-Ahead Improvement Phase

For the look-ahead method,the best packing result comes from the wall combination: left wall, right wall, far wall, upper wall and the parameter setting: $packing\_vol = 0.95, moving\_per = 1.0, vol\_per = 0.9, dim\_per = 0.1$. The average container volume utilization using this best parameter setting and wall combination can be increased to 0.87813642. The detailed packing result is listed in Figure 8.

Note that in the experimental results shown in Figure 8, the X-axis represents the test case index and the Y-axis represents the container volume utilization, and the average packing utilization and packing time are indicated on the top of each figure.

5 Comparison of Results

To show the effectiveness of the proposed packing algorithm and its coupled improvement strategy, we compare our results run on the test problems from the [OR-Lib,] with

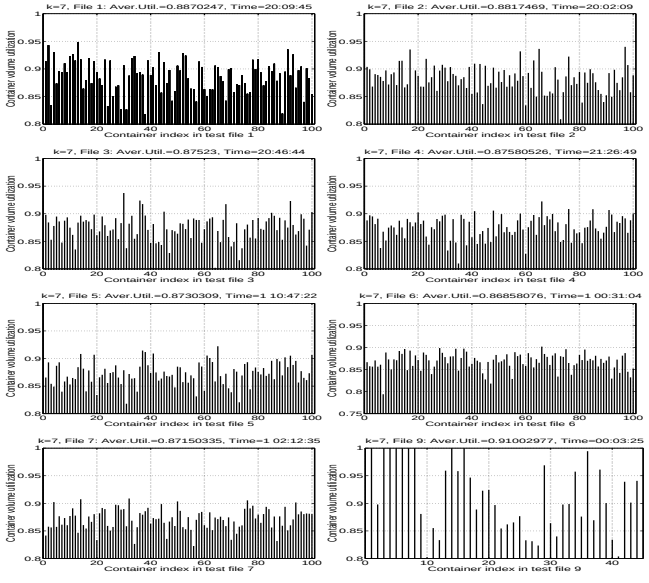

Figure 8: Best Experimental Result for Test File 1-7,9 Using Look-Ahead Improvement Phase

results in [E.E and Ratcliff, 1995] which is the only available work in the existing literature containing packing results on 700 test problems from the OR-Library.

	scheme(a)	scheme(b)	scheme(c)	MDBG Algo	LAM
file1	81.76%	83.79%	85.40%	83.31%	88.70%
file2	81.70%	84.44%	86.25%	84.26%	88.17%
file3	82.98%	83.94%	85.86%	84.61%	87.52%
file4	82.60%	83.71%	85.08%	84.39%	87.58%
file5	82.76%	83.80%	85.21%	84.44%	87.30%
file6	81.50%	82.44%	83.84%	84.61%	86.86%
file7	80.51%	82.01%	82.95%	84.63%	87.15%

Table 4: Performance Comparison Between the Existing Algorithms and the Proposed Algorithms

In Table 4, the scheme(a), scheme(b), scheme(c) are the respective scheme a, b and c in [E.E and Ratcliff, 1995]. MDBG Algo refers to the Multi-Directional Building-Growing Algorithm, and LAM refers to Look-Ahead Mechanism. From the experiment results, it is clear that the container volume utilization using the MDBG Algorithm is slightly better than that of scheme a and b in [E.E and Ratcliff, 1995]. But the MDBG Algorithm coupled with the LAM produces a much higher container volume utilization than scheme c in [E.E and Ratcliff, 1995]. The experimental results provide a strong evidence that the Multi-Directional Building-Growing Algorithm when coupled with the Look-Ahead mechanism gives an significant improvement to the container volume utilization for 3D container packing problem.

6 Conclusion

In this paper, we presented a new approach to solving the 3D container packing problem. Our paper is the first paper to benchmark our results across all the OR-Benchmarks. Our packing utilization of 87% compares very favorably against past works in this area.

References

[B.K. and K., 1993] Ngoi B.K. and Whybrew K. A fast spatial representation method. *Journal of International Advance Manufacturing Technology*, 8(2), 1993.

[E.E. and M.D., 1990] Bischoff E.E. and Marriott M.D. A comparative evaluation of heuristics for container loading. *European Journal of Operational Research*, 44:267–276, 1990.

[E.E and Ratcliff, 1995] Bischoff E.E and MSW Ratcliff. Issues in the development of approaches to container loading. *Omega. Int. J. Mgmt Sci.*, 23(4):337–390, 1995.

[Gehring H. and M., 1990] Menschner K. Gehring H. and Meryer M. A compute-based heuristic for packing pooled shipment containers. *European Journal of Operational Research*, 1(44):277–288, 1990.

[Han C.P. and J.P., 1989] Knott K. Han C.P. and Egbelu J.P. A heuristic approach to the three-dimensional cargo-loading problem. *Int. J. Prod. Res.*, 27(5):757–774, 1989.

[J.A and B.F., 1980] George J.A and Robinson B.F. A heuristic for packing boxes into a container. *Computer and Operational Research*, 7:147–156, 1980.

[K.A. and W.B., 1992] Dowsland K.A. and Dowsland W.B. Packing Problems. *European Journal of Operational Research*, 56:2–14, 1992.

[OR-Lib,] OR-Lib. http://www.ms.ic.ac.uk/library/.

[T.H. and Nee, 1992] Loh. T.H. and A.Y.C. Nee. A packing algorithm for hexahedral boxes. In *Proceedings of the Conference of Industrial Automation, Singapore*, pages 115–126, 1992.

[Wang, 2001] Ying Wang. 3D container packing. Honours thesis, National University of Singapore, 2001.

[W.B., 1991] Dowsland W.B. Three dimensional packing-solution approaches and heuristic development. *International Journal of Operational Research*, 29:1637–1685, 1991.

SEARCH, SATISFIABILITY, AND CONSTRAINT SATISFACTION PROBLEMS

SATISFIABILITY

Balance and Filtering in Structured Satisfiable Problems

Henry Kautz
Yongshao Ruan
Dept. Comp. Sci. & Engr.
Univ. Washington
Seattle, WA 98195
kautz@cs.washington.edu
ruan@cs.washington.edu

Dimitris Achlioptas
Microsoft Research
Redmond, WA 98052
optas@microsoft.com

Carla Gomes
Bart Selman
Dept. of Comp. Sci.
Cornell Univ.
Ithaca, NY 14853
gomes@cs.cornell.edu
selman@cs.cornell.edu

Mark Stickel
Artificial Intelligence Center
SRI International
Menlo Park, California 94025
stickel@ai.sri.com

Abstract

New methods to generate hard random problem instances have driven progress on algorithms for deduction and constraint satisfaction. Recently Achlioptas *et al.* (AAAI 2000) introduced a new generator based on Latin squares that creates only *satisfiable* problems, and so can be used to accurately test incomplete (one sided) solvers. We investigate how this and other generators are biased away from the uniform distribution of satisfiable problems and show how they can be improved by imposing a *balance* condition. More generally, we show that the generator is one member of a family of related models that generate distributions ranging from ones that are everywhere tractable to ones that exhibit a sharp hardness threshold. We also discuss the critical role of the problem encoding in the performance of both systematic and local search solvers.

1 Introduction

The discovery of methods to generate hard random problem instances has driven progress on algorithms for propositional deduction and satisfiability testing. Gomes & Selman (1997) introduced a generation model based on the quasigroup (or Latin square) completion problem (QCP). The task is to determine if a partially colored square can be completed so that no color is repeated in any row or any column. QCP is an NP-complete problem, and random instances exhibit a peak in problem hardness in the area of the phase transition in the percentage of satisfiable instances generated as the ratio of the number of uncolored cells to the total number of cells is varied. The structure implicit in a QCP problem is similar to that found in real-world domains: indeed, many problems in scheduling and experimental design take the form of a QCP. Thus, QCP complements earlier simpler generation models, such as random k-cnf (Mitchell *et al.* 1992). Like them QCP generates a mix of satisfiable and unsatisfiable instances.

In order to measure the performance of incomplete solvers, it is necessary to have benchmark instances that are known to be satisfiable. This requirement is problematic in domains where incomplete methods can solve larger instances than complete methods: it is not possible to use a complete method to filter out the unsatisfiable instances. It has proven difficult to create generators for satisfiable k-sat. Achlioptas *et al.* (2000) described a generation model for satisfiable quasigroup completion problems called "quasigroups with holes" (QWH). The QWH generation procedure basically inverts the completion task: it *begins* with a randomly-generated completed Latin square, and then erases colors or "pokes holes". The *backbone* of a satisfiable problem is the set of variables that receive the same value in all solutions to that problem. Achlioptas *et al.* (2000) showed that the hardest QWH problems arise in the vicinity of a threshold in the average size of the backbone.

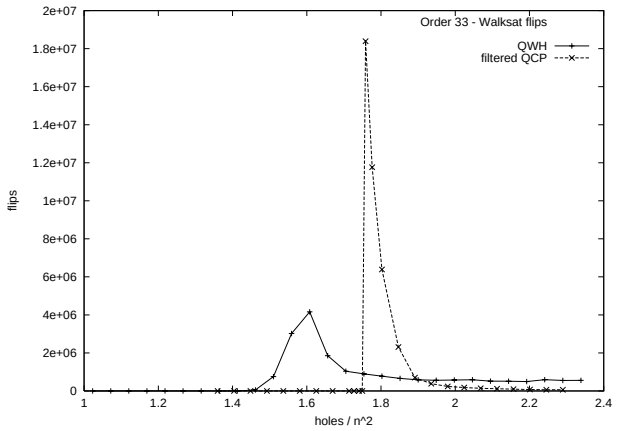

Figure 1: Comparison of the QWH and filtered QCP models for order 33 using Walksat. The x-axis is the percentage of holes (re-parameterized) and the y-axis is the number of flips.

model	Walksat flips	Satz backtracks	Sato branches
QWH	$4e+6$	41	$3e+5$
filtered QCP	$1e+8$	394	$1.5e+6$
balanced QWH	$6e+8$	4,984	$8e+6$

Table 1: Peak median hardness for different generation models for order 33 problems.

Despite the similarities between the filtered QCP and QWH models the problem distributions they generate are quite different. As noted by Achlioptas *et al.* (AAAI 2000), the threshold for QCP is at a higher ratio of holes than is the threshold for QWH. But even more significantly, we discovered that the hardest problems obtained using the filtered QCP

model are about an order of magnitude more difficult to solve than the hardest one obtained using the QWH model. Figure 1 illustrates the difference for the incomplete local search solver Walksat (Selman *et al.* 1992). (The critical parameter for a fixed order n is the percentage of holes (uncolored squares) in the problem. In all of the graphs in this paper the $x-$axis is re-parameterized as the number of holes divided by $n^{1.55}$. Achlioptas *et al.* (2000) demonstrates that this re-parameterization adjusts for different problem sizes.) The first two lines of Table 1 compare the peak hardness of QWH and filtered QCP for Walksat and two complete systematic solvers, Satz (Li & Anbulagan 1997) and Sato (Zhang 1997). These solvers are running on Boolean CNF encodings of the problem instances. The performance of such modern SAT solvers on these problems is competitive with or superior to the performance of optimized CSP solvers running on the direct CSP encoding of the instances (Achlioptas *et al.* 2000). (Note that the number of backtracks performed by Satz is usually much lower than the number performed by other systematic algorithms because of its use of lookahead.)

We begin by explaining this difference in hardness by showing how each generation model is *biased* away from the uniform distribution of all satisfiable quasigroup completion problems. Next, we introduce a new satisfiable problem model, *balanced QWH*, which creates problems that are even harder than those found by filtered QCP, as summarized in the last line of Table 1. This new model provides a tool for creating hard structured instances for testing incomplete solvers. In addition, its simplicity and elegance suggests that it will provide a good model to use for theoretical analysis of structured problems.

In the final section of the paper we turn to the issue of how we encode the quasigroup problems as Boolean satisfiability problems. For systematic methods the best encoding is based on a view of a quasigroup as a 3-dimensional cube, rather than as a 2-dimensional object. As shown below, the 2-D encodings are almost impossible to solve by systematic methods for larger orders. This difference is not unexpected, in that the 3-D encodings include more redundant constraints, which are known to help backtracking algorithms. For Walksat, however, a more complex picture emerges: the hardest instances can be solved more quickly under the 2-D encoding, while under-constrained instances are easier under the 3-D encoding.

2 The QWH and Filtered QCP Models

In order to better understand the QWH and filtered QCP models we begin by considering the problem of choosing a member uniformly at random from the set of all satisfiable quasigroup completion problems of order n with h uncolored cells. It is easy to define a generator for the uniform model: (1) Color $n^2 - h$ cells of an order n square randomly; (2) Apply a complete solver to test if the square can be completed; if not, return to step (1). This generator for the uniform model is, however, impractical for all but largest values of h. The problem is that the probability of creating a satisfiable instance in step (1) is vanishingly small, and so the generator will loop for an exponentially long time. In other words, the set of satisfiable quasigroup completion problems, although large, is small relative to the set of all completion problems.

Because the instances generated by placing down random colors are almost certainly unsatisfiable, the quasigroup completion generator from Gomes & Selman (1997) tries to filter out as many unsatisfiable configurations while incrementally partially coloring a square. The formulation is in terms of a CSP, with a variable for each cell, where the domain of the variable is initialized to the set of n colors. The generator repeatedly selects an uncolored cell at random, and assigns it a value from its domain. Then, forward-checking is performed to reduce the domains of cells that share a row or column with it. If forward-checking reaches an inconsistency (an empty domain) the entire square is erased and the process begins anew. Otherwise, once a sufficient number of cells have been colored arc consistency is checked; if the square passes this test, then it is output. The hardest mix of sat and unsat instances is generated when the number of holes is such that about 50% of the resulting squares are satisfiable. Because of it's popularity in the literature, we simply call the distribution generated by the process the QCP model. Adding a final complete solver to check satisfiability and eliminating those square which are unsatisfiable results in the filtered QCP model.[1]

The hardest hole ratio for the filtered QCP model is shifted toward the more constrained side. We discovered that hardest satisfiable instances occur when the filtering step eliminates about 90% of the partial Latin squares.

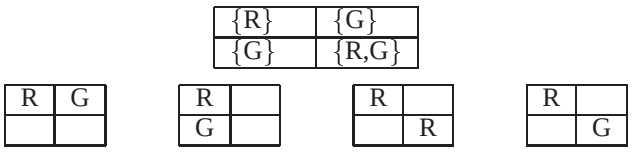

Figure 2: Top: the domains of each variable of an order 4 Latin square after the top left cell is colored R. Bottom: the four completion problems, generated with probabilities $1/3, 1/3, 1/6, 1/6$ respectively.

While the use of incremental forward checking successfully biases QCP away from unsatisfiable problems, it also has the unintended effect of introducing a bias between different *satisfiable* instances. For a simple example, consider generating order 2 problems with 2 holes, where the colors are R and G. Without loss of generality, suppose the first step of the algorithm is to color the top left square R. As shown in Figure 2, forward-checking reduces the domains of two of the remaining cells. The next step is to pick one of the three remaining cells at random, and then choose a color in its domain at random. As a result, one of the four partial Latin squares shown in the figure is created, but with different probabilities: the first two with probability $1/3$ each, and the last two with probability $1/6$ each. The fourth square is immediately eliminated by the final forward checking step. Because our choice of initial square and color was arbitrary, we see that each of the partial Latin squares where the two colored cells are in the same row or column is twice as like to be generated as one in which the two colored cells appear on a diagonal.

The QWH model introduced by Achlioptas *et al.* (2000) works as follows: (1) A random complete Latin square is cre-

[1]The QCP generator used in Gomes & Selman (1997) did not perform the final arc-consistency test. Thus, it generated a higher percentage of unsatisfiable instances. The filtered (satisfiable) distribution from that generator and ours are identical.

Figure 3: Two partial Latin squares with same pattern of holes but a different number of solutions, illustrating the bias of the QWH model.

ated by a Markov-chain process; (2) h random cells are uncolored. Does this generate the uniform model? Again the answer is no. Because all patterns of h holes are equally likely to be generated in step (2), we can without loss of generality consider both the number and pattern of holes to be fixed. The probability of creating a particular completion problem is proportional to the number of different complete Latin squares that yield that problem under the fixed pattern of holes. Therefore, we can show that QWH does not generate the uniform model by simply presenting two partial Latin squares with the same pattern of holes but a different number of solutions.

Such a counterexample appears in Figure 3: The square on the left has two solutions while the one on the right has only a single solution. In other words, the left square is twice as likely to be generated as the right square. In general, the QWH model is biased towards problems that have many solutions.

3 Patterns and Problem Hardness

We now consider different models of QCP and QWH, corresponding to different patterns. As we will see, the complexity of the models is dependent not only on the number of uncolored cells but also on the underlying pattern.

3.1 Rectangular and Aligned Models

In order to establish a baseline, we first consider two tractable models for QCP and QWH, which we refer to as *rectangular* and *aligned* models. Figure 4 (left and middle) illustrates such instances. In the *rectangular* QWH model, a set of columns (or rows) is selected and all the cells in these columns are uncolored. Moreover, one additional column (row) can be selected and be partially uncolored. The *aligned QWH* model is a generalization of the rectangular model in that we can pick both a set of rows and a set of columns and treat them as the chosen rows/columns in the rectangular model. Put differently, in the aligned model, the rows and columns can be permuted so that all cells in $C = \{1, \ldots, r-1\} \times \{1, \ldots, s-1\}$ are colored, some cells of $\{1, \ldots, r\} \times \{1, \ldots, s\} \setminus C$ are colored and all other cells are uncolored. We note that one could also naturally generate rectangular and aligned QCP instances.

In order to show that both the rectangular and aligned models are tractable, let us consider a Latin rectangle R on symbols $1, \cdots, n$. Let R(i) denote the number of occurrences of the symbol i in R, $1 \leq i \leq n$. We first introduce the following theorem from combinatorics.

Theorem: (Ryser 1951) An r x s Latin rectangle R on symbols $1, \cdots, n$ can be embedded in a Latin square of side n if and only if

$$R(i) \geq r + s - n \text{ for all } i, 1 \leq i \leq n.$$

Theorem: Completing a rectangular QCP or QWH is in P.

Proof: We can complete the rectangular QCP or QWH instance column by column (or row by row), starting with the column (or row) partially uncolored. (If there is no partially uncolored columns or rows, consider an arbitrary fully uncolored column or row.) We construct a bipartite graph G, with parts $U = \{u_1, u_2, \ldots, u_m\}$, and $W = \{w_1, w_2, \ldots, w_m\}$ in the following way: U represents the uncolored cells of the column (row) that needs to be colors; W represents the colors not used in that column (row). An edge (u_i,w_j) denotes that node u_i can be colored with color w_j. To complete the first column (row) we find a perfect matching in G. If there is no such matching we know that the instance is unsatisfiable. After completing the first column (row), we can complete the remaining columns (rows) in the same way.QED

Theorem: Completing an aligned QCP or QWH is in P.

Proof: We start by noting that an aligned instance can be trivially rearranged into an r x s rectangle, $s \leq n$, by permutating rows and columns (with possibly a row or column partially uncolored). So, if we show that we can complete an r x s rectangle into an r x n rectangle, we have proved our theorem, since we obtain an instance of the rectangular model. To complete an r x s rectangle into an r x n, again we complete a column (or row) at a time, until we obtain a rectangle of side n. For the completion of each column (or row), again, we solve a matching on the bipartite graph. The only difference resides in the way we build the graph: W only contains colors for which the condition in Ryser's theorem is satisfied. QED

3.2 Balance

While the rectangular and aligned models cluster holes, we now turn to a model that attempts to increase problem hardness by minimizing clustering. Let us start by reviewing the role of balance in the two most studied combinatorial optimization problems over random structures: random satisfiability and random graph coloring. It is particularly interesting to note the features shared by algorithms performing well on these problems.

Random k-SAT. A random formula is formed by selecting uniformly and independently m clauses from the set of all $2^k \binom{n}{k}$ k-clauses on a given set of n variables. The first algorithm to be analyzed on random k-SAT employs the *pure literal* heuristic repeatedly: a literal ℓ is satisfied only if $\bar{\ell}$ does not appear in the formula. Thus, a pure variable has all its occurrences appear with the same sign – a rather dramatic form of sign imbalance. The next key idea is unit-clause propagation, *i.e.*, immediately satisfying all clauses of length 1. More generally, dealing with shortest clauses first has turned out to be very useful. Subsequent improvements also come from exploiting imbalances in the formula: using degree information to determine which variable to set next and considering the number of positive and negative occurrences to determine value assignment.

Bayardo & Schrag (1996) gave experimental results on the role of balance, by considering random k-SAT formulas in which all literals occur in the same number of clauses. In such formulas, an algorithm has to first set a non-trivial fraction of all variables (essentially blindly) before any of the ideas mentioned above can start being of use. This suggests the potential of performing many more backtracks and indeed,

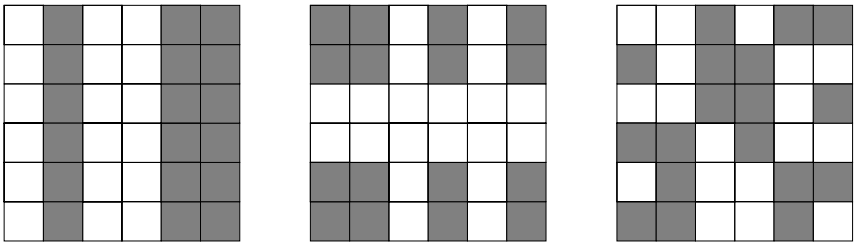

Figure 4: Left: an example of the rectangular model; middle: an example of the aligned model; right: a balanced model. Holes are in white.

balanced random k-SAT formulas are an order of magnitude harder than standard ones.

Random graph coloring. A random graph on n vertices is formed by including each of the $\binom{n}{2}$ edges with probability p. For graph coloring the most interesting range is $p = d/n$, where d is a constant. One can also consider *list-coloring*, where each vertex has a prescribed list of available colors. Analogously to the pure literal heuristic, a first idea is to exploit the advantage offered by low degree vertices. In particular, if we are using k colors then we can safely remove from the graph all vertices having degree smaller than k: if we can color the remaining graph then we can certainly complete the coloring on these vertices since each one of them will have at least one available color (as it has at most $k-1$ neighbors). Upon potentially reaching a k-core, where every vertex has degree at least k, it becomes useful to consider vertices having fewest available colors remaining and, among them, those of highest degree. Conversely, random graphs that are degree-regular and with all lists having the same size tend to be harder to color.

3.3 Balance in Random Quasigroups

In the standard QWH model we pick a random quasigroup and then randomly turn a number of its entries to holes. Fixing the quasigroup choice and the number of holes, let us consider the effect that different hole-patterns have on the hardness of the resulting completion problem. (For brevity, we only refer to rows below but naturally all our comments apply to columns just as well.)

Two extreme cases are rows with just one hole and rows with n holes. In the first case, the row can be immediately completed, while in the second case it turns out that given any consistent completion of the remaining $n \times (n-1)$ squares one can always complete the quasigroup. More generally, it seems like a good idea for any algorithm to attempt to complete rows having a smallest number of holes first, thus minimizing the branching factor in the search tree. Equivalently, having an equal number of holes in each row and column should tend to make things harder for algorithms.

Even upon deciding to have an equal number of holes in each row and column, the exact placement of the holes remains highly influential. Consider for example a pair of rows having holes only in columns i, j and further assume that there are no other holes in columns i, j. Then, by permuting rows and columns it is clear that we can move these four holes to, say, the top left corner of the matrix. Note now that in this new (isomorphic) problem the choices we make for these four holes are independent of all other choices we make in solving the problem, giving us a natural partition to two independent subproblems.

More generally, given the 0/1 matrix A where zeros correspond to colored entries and ones correspond to holes, one can attempt to permute the rows and columns to minimize the *bandwidth* of A (the maximum absolute difference between i and j for which $A(i,j)$ is 1). Having done so, the completion problem can be solved in time exponential in the bandwidth.

We were surprised to discover that even though Satz contains no code that explicitly computes or makes use of bandwidth (indeed, *exactly* computing the bandwidth of a problem is NP-complete), it is extremely efficient for bounded-bandwidth problems. We generated bounded-bandwidth instances by first punching holes in a band along the diagonal of a Latin square, and then shuffling the rows and columns 1,000 times to hide the band. Satz solved all instances of this type for all problem sizes (up to the largest tested, order 33) and hole ratios in either 0 or 1 backtrack! Satz's strategy is Davis-Putnam augmented with one-step lookahead: this combination is sufficient to uncover limited-bandwidth instances.

When the holes are placed randomly then with high probability the resulting matrix A will have high bandwidth. Alternatively, we can view A as a random $n \times n$ bipartite graph in which vertex i is connected to vertex j iff there is a hole in position (i, j). The high bandwidth of A then follows from relatively standard results from random graph theory (Fernandez de la Véga 1981). Moreover, having an equal number of holes in each row/column makes the random graph regular, which *guarantees* that A has large bandwidth. Intuitively, small balanced instances should exhibit properties that hold of large random instances in the asymptotic limit.

Viewing the hole pattern as a regular random bipartite graph readily suggests a way for generating a uniformly random hole pattern with precisely q holes in each row and column for any q (*i.e.*, $q = h/n$) by the following algorithm:

Set $H = \emptyset$.
Repeat q times:
  Set T to $\{1, \ldots, n\} \times \{1, \ldots, n\} \setminus H$.
  Repeat n times:
    Pick a uniformly random $(i, j) \in T$;
    Add (i, j) to H;
    Remove all elements in row i and column j from T.

Balancing can be applied to either the QWH or QCP models: in the former, the pattern is used to uncolor cells; in the latter, the pattern is used to determine the cells that are *not* colored incrementally.

3.4 Empirical Results

We measured the difficulty of solving problem distributions generated under each of the models using three different al-

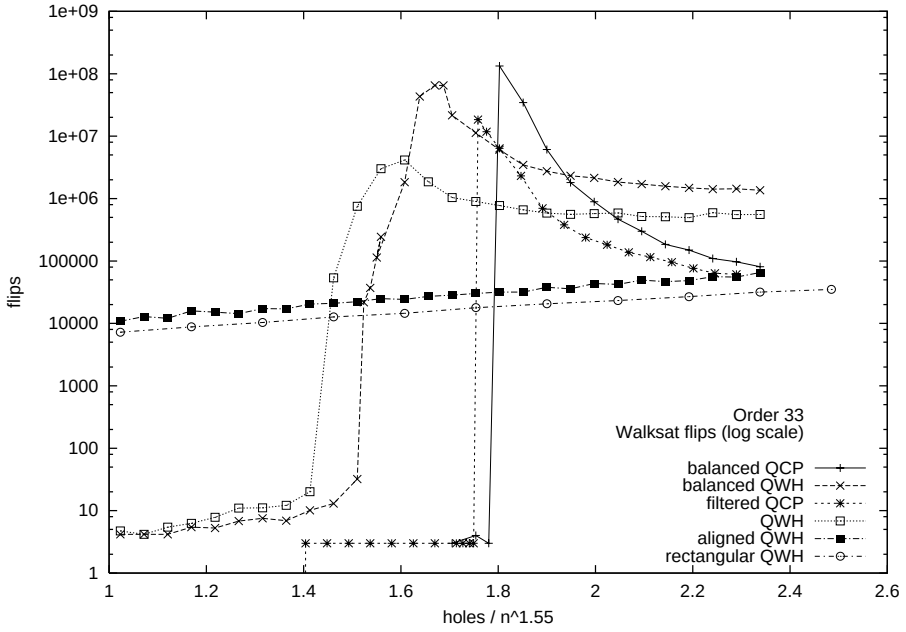

Figure 5: Comparison of generation models for order 33 using Walksat. Y-axis is the number of flips (log scale) and x-axis is the percentage of holes (re-parameterized). Each data point represents the median number of flips for 100 instances. Less-constrained instances appear to the right.

gorithms. As a preliminary step problems were simplified by arc consistency and then translated into SAT problems in conjunctive normal form, as described in Sec. 4 below. For each data point for a range of percentage of holes 100 problems were generated and solved by three different algorithms: Walksat, which performs local search over the space of truth assignments, using 30% noise and no cutoff; Satz, which implements the backtracking Davis-Putnam algorithm augmented with 1-step lookahead; and Sato (Zhang 1997), another Davis-Putnam type solver that uses dependency-directed backtracking and clause learning. In this paper we present the data for order 33, a size which clearly distinguishes problem difficulty but still allows sufficient data to be gathered. For reasons of space, we will omit detailed results for Sato, which are qualitatively similar to those for Satz.

Figure 5 displays the results for Walksat: note the log scale, so that each major division indicates an *order of magnitude* increase in difficulty. *Less* constrained problems appear to the *right* in the graphs: note that is the opposite of the convention for presenting results on random k-cnf (but is consistent with Achlioptas *et al.* (2000)). On the log scale we see that the most constrained instances are truly easy, because they are solved by forward-checking alone. The under-constrained problems are of moderate difficulty. This is due to the fact that the underconstrained problem are simply much larger after forward-checking. Again note that this situation is different from that of random k-cnf, where it is the over-constrained problems that are of moderate difficulty (Cook and Mitchell 1997).

At the peak the balanced QWH problems are much harder than the filtered QCP problems, showing that we have achieved the goal of creating a better benchmark for testing incomplete solvers. Balancing can be added to the filtered QCP model; the resulting balanced QCP model are yet more difficult. This indicates that balancing does not make QWH and QCP equivalent: the biases of the two approaches remain distinct. Both of the QWH models are harder than the QCP models in the under-constrained region to the right; we do not yet have an explanation for this phenomena.

Both the aligned and rectangle models are easy for Walksat, and show no hardness peak. In the over-constrained area (to the left) they require more flips to solve than the others. This is because clustering all the holes prevents arc consistency from completely solving the problems in the over-constrained region (there are more than one solutions), as it usually does for the other models.

Figure 6 shows the same ordering of hardness peaks for Satz. The behavior of Satz on the rectangle case is an interesting anomaly: it quickly becomes lost on the under-constrained problems and resorts to exhaustive search. This is because Satz gains its power from lookahead, and on under-constrained rectangular problems lookahead provides no pruning. Sato, which employs look-back rather than lookahead, does not exhibit this anomaly: it solves the rectangular problems as easily as the aligned ones.

We also measured the variance in the number of holes per row or column and the bandwidth of the balanced and random models. As expected, the variance was very low for the balanced case, averaging between 0.0 and 0.2 over the rage of ratios, compared with a range of 5.4 to 8.2 for the random case. Thus non-balanced problems will often have rows or columns that contain only a few, highly-constrained holes that can be easily filled in.

4 3-D and 2-D Encodings

Up to this point of the paper we have been concerned with understanding what makes a problem *intrinsically* hard. In practice, the difficulty of solving a particular instance using a particular algorithm is also dependent upon the details of the

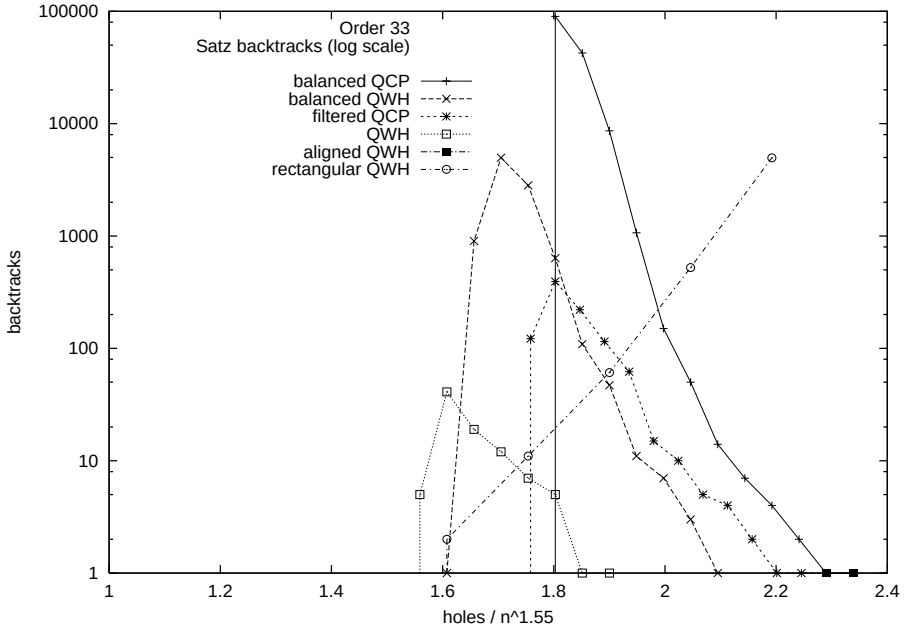

Figure 6: Comparison of generation models for order 33 using the Davis-Putnam type solver Satz. The x-axis is the percentage of holes (re-parameterized) and the y-axis is the number of backtracks (log scale). Each data point represents the median number of backtracks for 100 instances.

holes	holes/$n^{1.55}$	2-D	3-D
247	1.41	18	17
254	1.45	33	66
261	1.49	32	108
268	**1.53**	**23**	**109**
275	1.57	17	61
282	1.61	14	61
289	1.65	12	23

Table 2: Average number unit propagations performed immediately after each branch by Satz for the 2-D and 3-D encodings of order 28 QWH instances. Hardness peak is at $n^{1.55} = 1.53$ (261 holes).

representation of the problem.

Although quasigroup completion problems are most naturally represented as a CSP using multi-valued variables, encoding the problems using only Boolean variables in clausal form turns out to be surprisingly effective. Each Boolean variable represents a color assigned to a cell, so where n is the order there are n^3 variables. The most basic encoding, which we call the "2-dimensional" encoding, includes clauses that represent the following constraints:

1. Some color must be assigned to each cell;
2. No color is repeated in the same row;
3. No color is repeated in the same column.

Constraint (1) becomes a clause of length n for each cell, and (2) and (3) become sets of negative binary clauses. The total number of clauses is $O(n^4)$.

The binary representation of a Latin square can be viewed as a cube, where the dimensions are the row, column, and color. This view reveals an alternative way of stating the Latin square property: any set of variables determined by holding two of the dimensions fixed must contain exactly one true variable. The "3-dimensional" encoding captures this condition by also including the following constraints:

1. Each color much appear at least once in each row;
2. Each color much appear at least once in each column;
3. No two colors are assigned to the same cell.

As before, the total size of the 3-D encoding is $O(n^4)$.

As reported in Achlioptas *et al.* (2000), state of the art backtracking and local search SAT solvers using the 3-D encoding are competitive with specialized CSP algorithms. This is particularly surprising in light of the fact that the best CSP algorithms take explicit advantage of the structure of the problem, while the SAT algorithms are generic. Previous researchers have noted that the performance of backtracking CSP solvers on quasigroup problems is enhanced by using a dual representation (Slaney *et al.* 1995, Shaw *et al.* 1998, Zhang and Stickel 2000). This suggests a reason for the success of Davis-Putnam type SAT solvers: In the CSP dual encoding, there are variables for color/row pairs, where the domain is the set of columns, and similarly for color/column pairs, where the domain is the set of rows. The 3-D SAT encoding essentially gives us these dual variables and constraints for free.

This explanation is supported by the extremely poor performance of SAT solvers on the 2-D encodings of the problems. Neither Satz nor Sato can solve any instances at the hardness peak for orders larger than 28; using the 3-D encoding, by contrast, either could solve all instances with one backtrack on average. As shown in Figure 7, the work required by Satz explodes as the problem becomes underconstrained , requiring over 100,000 backtracks for order 28.

An explanation for the difference in performance of Satz on the different encodings can be found by examining the number of unit propagations triggered by each split in the search trees. Table 2 compares the number of unit propagations around the point at which the 2-D encodes become hard

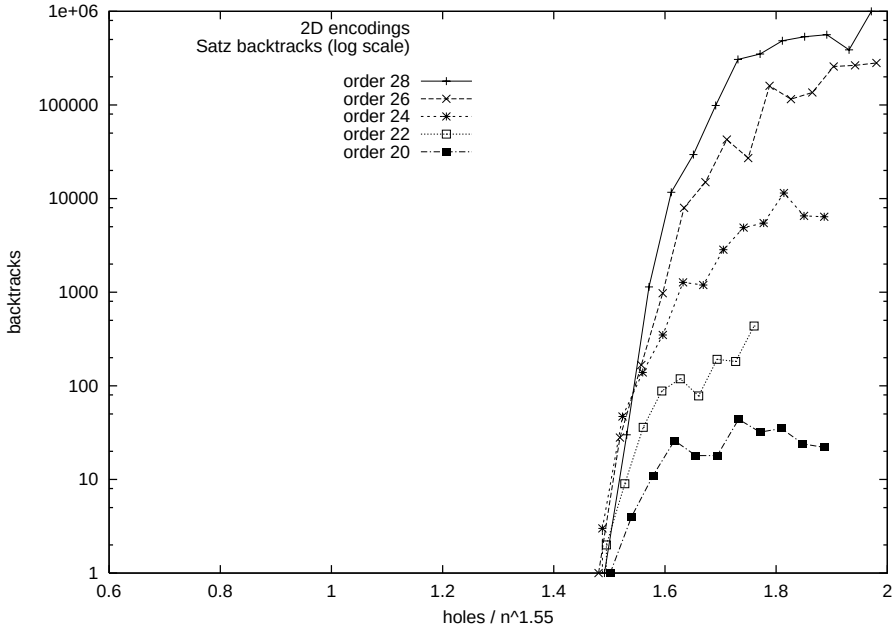

Figure 7: Comparison of 3-D versus 2-D encodings using Satz (backtracks, log scale) for 2-D encodings for orders 20 to 28. All problems of this size using the 3-D encodings could be solved by Satz in 0 or 1 backtracks.

for Satz (in bold). Note that at this point each split sets about 5 times more variables for the 3-D encoding than for the 2-D encodings.

Are the 2-D encodings inherently hard? Consider the performance of Walksat, on even larger orders (30 and 33), shown on in Figure 8. Walksat shows an unusual pattern: the 2-D encodings are somewhat *easier* than the 3-D encodings at the peak, and somewhat harder than then 3-D encodings in the under-constrained region to the right. Thus the 2-D and 3-D are in fact incomparable in terms of any general notion of hardness.

A significant difference between the 3-D and 2-D encodings is that for both Walksat and Satz it is difficult to see any hardness peak at the threshhold: the problems become hard and then stay at least as hard as they become more and more under-constrained. Note that the most underconstrained instances are *inherently* easy, since they correspond to a *empty* completion problem. This reinforces our argument that the 3-D encoding more accurately reflects the underlying computational properties of the quasigroup problem.

In summary, it is important to distinguish properties of a problem instance that make it inherently hard for all methods of attack and properties that make it accidently hard for particular methods. While the encoding style is such an accidental property, the main conjecture we present in this paper is that balance is an inherent property. The evidence for the conjecture is that increasing balance increases solution time for a *variety* of solvers.

5 Conclusions

Models of random problem generation serve two roles in AI: first, to provide tools for testing search and reasoning algorithms, and second, to further our understanding of what makes *particular* problems hard or easy to solve, as distinct from the fact that they fall in a class that is *worst-case* intractable. In this paper we introduced a range of new models of the quasigroup completion problem that serve these roles. We showed how a new notion of balance is an important factor in problem hardness. While previous work on balancing formulas considered the roles of positive and negative literals, our notion of balance is purely *structural*. Balancing improves the usefulness of the QWH model – one of the best models known for testing incomplete solvers – while retaining its formal simplicity and elegance.

References

D. Achlioptas, C. Gomes, H. Kautz, B. Selman (2000). Generating Satisfiable Instances. *Proc. AAAI-2000*.

Bayardo, R.J. and Schrag, R.C. (1996) Using csp look-back techniques to solve exceptionally hard sat instances. *Proc. CP-96)*, 46–60.

Cheeseman, P. and Kanefsky, R. and Taylor, W. (1991). Where the Really Hard Problems Are. *Proc. IJCAI-91*, 163–169.

Cook, S.A. and Mitchell, D. (1997). Finding Hard Instances of the Satisfiability Problem: A Survey, in D. Du, J. Gu, and P. Pardalos, eds. *The Satisfiability Problem.* Vol. 35 of DIMACS Series in Discr. Math. and Theor. Comp. Sci., 1-17.

Fernandez de la Véga, W. (1981) On the bandwidth of random graphs. *Combinatorial Mathematics*, North-Holland, 633–638.

Gent, I. and Walsh, T. (1993) An empirical analysis of search in GSAT. *J. of Artificial Intelligence Research*, vol. 1, 1993.

Gibbs, N.E. , Poole, Jr., W.E. and Stockmeyer, P.K. (1976). An algorithm for reducing the bandwidth and profile of a sparse matrix, *SIAM J. Numer. Anal.*, 13 (1976), 236–249.

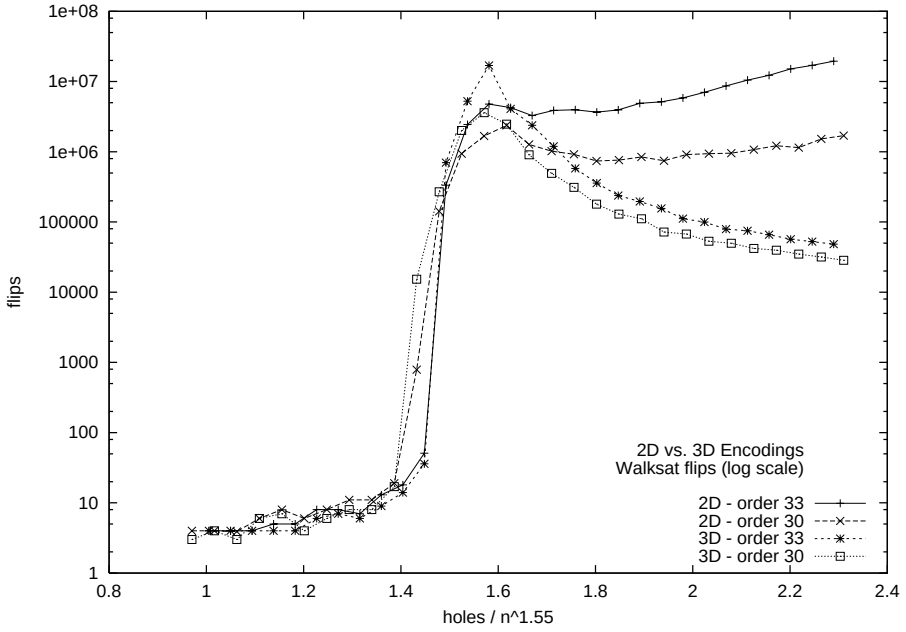

Figure 8: Comparison of 3-D versus 2-D encodings using Walksat (flips, log scale) for orders 30 and 33.

Gomes, C.P. and Selman, B. (1997a). Problem structure in the presence of perturbations. *Proc. AAAI-97*.

Hall, M. (1945). An existence theorem for latin squares. *Bull. Amer. Math Soc.*, 51, (1945), 387–388.

Hogg, T., Huberman, B.A., and Williams, C.P. (Eds.) (1996). Phase Transitions and Complexity. *Art. Intell.*, 81, 1996.

Hoos, H. 1999. SATLIB. A collection of SAT tools and data. See www.informatik.tu-darmstadt.de/AI/SATLIB.

Impagliazzo, R., Levin, L., and Luby, M. (1989). Pseudo-random number generation of one-way functions. *Proc. 21st STOC*.

Kirkpatrick, S. and Selman, B. (1994). Critical behavior in the satisfiability of Boolean expressions. *Science*, 264, 1994, 1297–1301.

Li, Chu Min and Anbulagan (1997). Heuristics based on unit propagation for satisfiability problems. *Proc. IJCAI-97*, 366–371.

Mitchell, D., Selman, B., and Levesque, H.J. (1992). Hard and easy distributions of SAT problems. *Proc. AAAI-92*, 459–465.

Regin, J.C. (1994). A filtering algorithm for constraints of difference in CSP. *Proc. AAAI-94*, 362–367.

Ryser, H. (1951). A combinatorial theorem with an application to latin rectangles. *Proc. Amer. Math. Soc.*, 2, (1951), 550-552.

Selman, B. and Levesque, H.J., and Mitchell, D.G. (1992). A New Method for Solving Hard Satisfiability Problems. *Proc. AAAI-92*.

Shaw, P., Stergiou, K., and Walsh, T. (1998) Arc consistency and quasigroup completion. *Proc. ECAI-98*, workshop.

Slaney, J., Fujita, M., and Stickel, M. (1995) Automated reasoning and exhaustive search: quasigroup existence problems. *Computers and Mathematics with Applications*, 29 (1995), 115-132.

Stergiou, K. and Walsh, T. (1999) The Difference All-Difference Makes. Proc. of *IJCAI-99*.

Van Gelder, A. (1993). Problem generator (mkcnf.c) contributed to the DIMACS 1993 Challenge archive.

Zhang, H. (1997). SATO: An Efficient Propositional Prover. Proc. CADE-97.

Zhang, H. and Stickel, M.E. (2000) Implementing the Davis-Putnam method. *Journal of Automated Reasoning* 24(1-2) 2000, 277-296.

Efficient Consequence Finding

Laurent Simon
Laboratoire de Recherche en Informatique
U.M.R. CNRS 8623, Université Paris-Sud
91405 Orsay Cedex, France
simon@lri.fr

Alvaro del Val
E.T.S. Informática, B-336
Universidad Autónoma de Madrid
28049 Madrid, Spain
delval@ii.uam.es

Abstract

We present an extensive experimental study of consequence-finding algorithms based on kernel resolution, using both a trie-based and a novel ZBDD-based implementation, which uses Zero-Suppressed Binary Decision Diagrams to concisely store and process very large clause sets. Our study considers both the full prime implicate task and applications of consequence-finding for restricted target languages in abduction, model-based and fault-tree diagnosis, and polynomially-bounded knowledge compilation. We show that the ZBDD implementation can push consequence-finding to a new limit, solving problems which generate over 10^{70} clauses.

1 Introduction

Many tasks in Artificial Intelligence can be posed as consequence-finding tasks, i.e. as the problem of finding certain consequences of a propositional knowledge base. These include prime implicates, abduction, diagnosis, nonmonotonic reasoning, and knowledge compilation. [Marquis, 1999] provides an excellent survey of the field and its applications. Unfortunately, there have been very few serious computational attempts to address these problems. After some initial excitement with ATMSs [de Kleer, 1986] and clause management systems [Reiter and de Kleer, 1987; Kean and Tsiknis, 1993], it was soon realized that these systems would often run out of resources even on moderately sized instances, and interest on the topic waned. Thus, for example, [Marquis, 1999] says in concluding his survey:

> "The proposed approaches [to consequence-finding] are however of limited computational scope (algorithms do not scale up well), and there is only little hope that significantly better algorithms could be designed"

In this paper, we conduct the first extensive experimental study of consequence-finding (CF), more specifically of one of the approaches referred to in the above quote, namely kernel resolution [del Val, 1999]. Kernel resolution allows very efficient focusing of consequence-finding on a subset of "interesting" clauses, similar to SOL resolution [Inoue, 1992]. We also introduce novel consequence-finding technology, namely, the use of ZBDDs (Zero-Suppressed Binary Decision Diagrams [Minato, 1993]) to compactly encode and process extremely large clause sets. As a result of this new technology, the well-known Tison method [Tison, 1967] for finding the prime implicates of a clausal theory, a special case of kernel resolution, can now efficiently handle more than 10^{70} clauses. The combination of focused consequence-finding with ZBDDs makes CF able to deal computationally with significant instances of the applications discussed above for the first time. In particular, our experiments include both full CF (the prime implicate task), and applications of consequence-finding for restricted target languages in abduction, model-based and fault-tree diagnosis, and polynomially-bounded knowledge compilation.

We assume familiarity with the standard literature on propositional reasoning and resolution. Some definitions are as follows. A clause C subsumes a clause D iff $C \subseteq D$. The empty clause is denoted $\Box$. For a theory (set of clauses) Σ, we use $\mu(\Sigma)$ to denote the result of removing all subsumed clauses from Σ. An implicate of Σ is a clause C such that $\Sigma \models C$; a prime implicate is an implicate not subsumed by any other implicate. We denote by $PI(\Sigma)$ the set of prime implicates of Σ. We are often interested only in some subset of $PI(\Sigma)$. For this purpose, we define the notion of a target language $\mathcal{L}_T$, which is simply a set of clauses. We assume $\mathcal{L}_T$ is *closed under subsumption (c.u.s.)*, i.e. for any $C \in \mathcal{L}_T$ and $D \subseteq C$, we have $D \in \mathcal{L}_T$. A target language can always be closed under subsumption by adding all subsumers of clauses in the language.

Given these definitions, the task we are interested in is finding the prime $\mathcal{L}_T$-implicates of Σ, defined as $PI_{\mathcal{L}_T}(\Sigma) = PI(\Sigma) \cap \mathcal{L}_T$. We will mainly consider the following target languages: $\mathcal{L}$ is the full language, i.e. the set of all clauses over the set $Var(\Sigma)$ of variables of Σ. $\mathcal{L}_\Box = \{\Box\}$ contains only the empty clause. Given a set of variables V, the "vocabulary-based" language $\mathcal{L}_V$ is the set of clauses over V. Finally, for a constant K, $\mathcal{L}_K$ is the set of clauses over $Var(\Sigma)$ whose length does not exceed K. Thus we have $\mathcal{L}_1$, $\mathcal{L}_2$, etc.

Each of these languages corresponds to some important AI task. At one extreme, finding the prime implicates of Σ is simply finding $PI_{\mathcal{L}}(\Sigma) = PI(\Sigma)$; at the other extreme, deciding whether Σ is satisfiable is identical to deciding whether

$PI_{\mathcal{L}_\Box}(\Sigma)$ is empty. Vocabulary-based languages also have many applications, in particular in abduction, diagnosis, and non-monotonic reasoning (see e.g. [Inoue, 1992; Selman and Levesque, 1996; Marquis, 1999] among many others). Finally, $\mathcal{L}_K$ or subsets thereof guarantee that $PI_{\mathcal{L}_K}(\Sigma)$ has polynomial size, which is relevant to knowledge compilation (surveyed in [Cadoli and Donini, 1997]).

Sometimes we will be interested in theories which are logically equivalent to $PI_{\mathcal{L}_T}(\Sigma)$, but which need not include all $\mathcal{L}_T$-implicates, and can thus be much more concise. We refer to any such theory as a $\mathcal{L}_T$-LUB of Σ, following [Selman and Kautz, 1996], see also [del Val, 1995]. We'll see one particular application of $\mathcal{L}_T$-LUBs in our discussion of diagnosis later.

2 Kernel resolution: Review

Kernel resolution, described in [del Val, 1999; 2000a], is a consequence-finding generalization of ordered resolution. We assume a total order of the propositional variables $x_1, \ldots, x_n$. A *kernel clause* C is a clause partitioned into two parts, the skip $s(C)$, and the kernel $k(C)$. Given any target language $\mathcal{L}_T$ closed under subsumption, a *$\mathcal{L}_T$-kernel resolution deduction* is any resolution deduction constructed as follows: (a) for any input clause C, we set $k(C) = C$ and $s(C) = \emptyset$; (b) resolutions are only permitted upon kernel literals; (c) the literal l resolved upon partitions the literals of the resolvent into those smaller (the skip), and those larger (the kernel) than l, according to the given ordering; and (d) to achieve focusing, we require any resolvent R to be $\mathcal{L}_T$-acceptable, which means that $s(R) \in \mathcal{L}_T$.

In order to search the space of kernel resolution proofs, we associate to each variable x_i a bucket $b[x_i]$ of clauses containing x_i. The clauses in each bucket are determined by an indexing function $I_{\mathcal{L}_T}$, so that $C \in b[x_i]$ iff $x_i \in I_{\mathcal{L}_T}(C)$. We can always define $I_{\mathcal{L}_T}(C) = \{$kernel variables of the largest prefix $l_1 \ldots l_k$ of C s.t. $l_1 l_2 \ldots l_{k-1} \in \mathcal{L}_T\}$, where C is assumed sorted in ascending order [del Val, 1999]; resolving on any other kernel literal would yield a non-$\mathcal{L}_T$-acceptable resolvent.

Bucket elimination, abbreviated $\mathcal{L}_T$-BE, is an exhaustive search strategy for kernel resolution. $\mathcal{L}_T$-BE processes buckets $b[x_1], \ldots, b[x_n]$ in order, computing in step i all resolvents that can be obtained by resolving clauses of $b[x_i]$ upon x_i, and adding them to their corresponding buckets, using $I_{\mathcal{L}_T}$. We denote the set of clauses computed by the algorithm as $\mathcal{L}_T$-BE(Σ). The algorithm, which uses standard subsumption policies (so that $\mathcal{L}_T$-BE$(\Sigma) = \mu(\mathcal{L}_T$-BE$(\Sigma))$), is complete for finding consequences of the input theory which belong to the target language $\mathcal{L}_T$, that is, $\mathcal{L}_T$-BE$(\Sigma) \cap \mathcal{L}_T = PI_{\mathcal{L}_T}(\Sigma)$.

As shown in [del Val, 1999], $\mathcal{L}$-BE is identical to Tison's prime implicate algorithm [Tison, 1967], whereas $\mathcal{L}_\Box$-BE is identical to directional resolution (DR), the name given by [Dechter and Rish, 1994] to the original, resolution-based Davis-Putnam satisfiability algorithm [Davis and Putnam, 1960]. $\mathcal{L}_\Box$-BE(Σ) is called the *directional extension* of Σ, and denoted $DR(\Sigma)$.

For $\mathcal{L}_V$ we will in fact consider two BE procedures, both of which assume that *the variables of V are last in the ordering*. $\mathcal{L}_V^1$-BE is simply $\mathcal{L}_V$-BE under this ordering assumption. $\mathcal{L}_V^0$-BE is identical, except that processing is interrupted right before the first variable of V is processed. Thus $\mathcal{L}_V^1$-BE$(\Sigma) \cap \mathcal{L}_V = PI_{\mathcal{L}_V}(\Sigma)$, whereas $\mathcal{L}_V^0$-BE$(\Sigma) \cap \mathcal{L}_V$ is logically equivalent but not necessarily identical to $PI_{\mathcal{L}_V}(\Sigma)$; i.e., it is a $\mathcal{L}_V$-LUB of Σ. Note that, in either case, the desired set of clauses is stored in the last buckets. The advantage of this ordering is that either form of $\mathcal{L}_V$-BE behave exactly as directional resolution, a relatively efficient *satisfiability* method, up to the first V-variable of the ordering. $\mathcal{L}_V^0$-BE stops right there (and is thus strictly cheaper than deciding satisfiability with DR under such orderings), while $\mathcal{L}_V^1$-BE continues, computing the prime implicates of $\mathcal{L}_V^0$-BE$(\Sigma) \cap \mathcal{L}_V$ with full kernel resolution over the V-buckets.

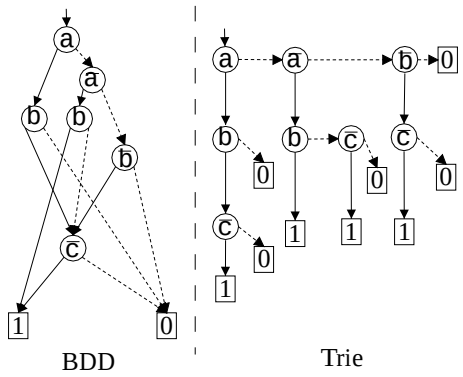

Figure 1: ZBDD and trie representation for the set of clauses $\Sigma = \{a \vee b \vee \neg c, \neg a \vee b, \neg a \vee \neg c, \neg b \vee \neg c\}$.

3 Zero–Suppressed BBDs: Review

[Bryant, 1992] provides an excellent survey and introduction of Binary Decision Diagrams (BDD). A BDD is a directed acyclic graph with a unique source node, only two sinks nodes (1 and 0, interpreted respectively as $true$ and $false$) and with labeled nodes $\Delta(x, n1, n2)$. Such node x has only two children (n_1 and n_2, connected respectively to its 1-arc and 0-arc) and is classically interpreted as the function $f = if\, x\, then\, f_1\, else\, f_2$ (if f_1 and f_2 interpret the BDDs n_1 and n_2).

The power of BDDs derives from their reduction rules. A ROBDD (Reduced Ordered BDD, simply noted BDD in the following) requires that its variables are sorted according to a given order and that the graph does not contain any isomorphic subgraph (*node-sharing* rule). In addition, the *node-elimination* rule deletes all nodes $\Delta(x, n, n)$ that do not care about their values. With the classical semantics, each node is labeled by a variable and each path from the source node to the 1-sink represents a model of the encoded formula. The BDD can thus be viewed as an efficient representation of a Shannon normal tree.

In order to use the compression power of BDDs for encoding sparse sets instead of just boolean functions, [Minato, 1993] introduced Zero-Suppressed BDDs (ZBDDs). Their principle is to encode the boolean characteristic function of a

set. For this purpose, Minato changed the *node-elimination* rule into the *Z-elimination* rule for which useless nodes are those of the form $\Delta(x, 0, n)$. So, if a variable does not appear on a path, then its default interpretation is now *false* (which means *absent*, instead of *don't care*). If one wants to encode only sets of clauses, each ZBDD variable needs to be labeled by a literal of the initial formula, and each path to the 1-sink now represents the clause which contains only the litterals labeling the parents of all 1-arcs of this path. Figure 1 represents the ZBDD encoding the set of clauses $\Sigma = \{a \vee b \vee \neg c, \neg a \vee b, \neg a \vee \neg c, \neg b \vee \neg c\}$.

The "compressing" power of ZBDDs is illustrated by the fact that there exist theories with an exponential number of clauses that can be captured by ZBDDs of polynomial size. Because their complexity only depends on the size of the ZBDD, not on the size of the encoded set, ZBDD operators can be designed to efficiently handle sets of clauses of an exponential size.

4 Kernel resolution on ZBDDs

In order to deal with this compact encoding in the context of kernel resolution, we need to define a way to efficiently obtain resolvents, and to identify buckets. For the former task, [Chatalic and Simon, 2000b; 2000c] introduced a very efficient *multiresolution rule* which works directly on sets of clauses, and thus which can compute the result of eliminating a variable without pairwise resolving clauses.

Definition (multiresolution): Let Δ^+ and Δ^- be two sets of clauses without the variable x_i. Let $\mathcal{R}_i$ be the set of clauses obtained by distributing the set Δ^+ over Δ^-. Multiresolution is the following rule of inference:

$$(x_i \vee \bigwedge \Delta^+) \wedge (\neg x_i \vee \bigwedge \Delta^-) \Rightarrow \bigwedge \mathcal{R}_i.$$

A bucket $b[x_i]$ can always be expressed in the form required by this rule, so that $\mathcal{R}_i$ corresponds to the set of resolvents obtained by processing $b[x_i]$. The main advantage of this definition is that it can be shown that if $\mathcal{R}_i$ is computed directly at the set level, without explicitly enumerating clauses, its complexity can be independent of the number of classical resolution steps. [Chatalic and Simon, 2000b] propose a system called ZRes which implements the multiresolution principle by means of ZBDDs. ZRes manipulates clause sets directly, through their ZBDD representation, with no explicit representation of individual clauses. A bucket is easily retrieved from the ZBDD storing the current clause set in the form of ZBDDs for appropriate Δ_i^+ and Δ_i^-; and it is then processed using a specialized ZBDD operator, *clause-distribution*, which implements multiresolution by distributing Δ_i^+ over Δ_i^- to obtain $\mathcal{R}_i$. The system comes with other ZBDDs operators designed for clause-set manipulation, such as *subsumption-free union* and *set-difference*. All these three operators delete subsumed and tautologous clauses as soon as possible, during the bottom-up construction of the resulting ZBDD.

To generalize these ideas for focused kernel resolution, we need to deal with complex indexing functions. For this purpose, we keep two ZBDDs, for dead and live clauses respectively. A "dead" clause is one which can no longer be resolved according to the $\mathcal{L}_T$-restriction in place, given the variables that have been processed so far. For example, a clause becomes dead for $\mathcal{L}_\square$ (i.e. for DR) when just one of its variables is processed; and for $\mathcal{L}_K$ when its first $K+1$ variables are processed. Right before processing a variable, we obtain "its bucket" from the live ZBDD. Processing the variable is then done by multiresolution. Finally, when done with a variable, new clauses become dead, and they are moved from the live ZBDD to the dead ZBDD.

5 Experimental results

Our experimental tests, which include structured benchmark gathering and generating as well as random instances testing, represents more than 60 cpu-days on the reference machine[1], more than 10000 runs on structured examples and almost the same on random instances. Only selected results are presented here, for an obvious lack of space.

Times are reported in seconds. All experiments use a 1000 seconds timeout, and a dynamic min-diversity heuristic for variable ordering. This heuristic chooses the variable for which the product of its numbers of positive and negative occurrences is minimal, thus trying to minimize the number of resolvents.

In addition to the ZBDD-based implementations, labeled `zres`, we also consider algorithms based on the trie-data structure [de Kleer, 1992], namely, `picomp` and `drcomp`, for respectively prime implicates and directional resolution. In contrast to the ZBDD programs, trie-based programs explicitly represent clauses and buckets, and use pairwise resolution. The trie (illustrated in Figure 1) includes a number of significant optimizations to minimize trie traversal, for efficient subsumption.

It can be shown that ZBDDs are never worse than tries in space requirements, and can be much better. Intuitively, while tries can only share common structure on the prefix of clauses, ZBDDs can do it on their prefix and postfix, i.e. they allow factoring clauses from both the beginning and the end. In terms of time, as we will see, the situation is more complex. Nevertheless, the ZBDD implementation is the one that really pushes consequence-finding well beyond its current limits.

5.1 Prime implicates

Prime implicate computation remains one of the fundamental consequence finding tasks. We focus in this section on our two very different Tison's algorithm implementations, `picomp` and `zres-tison`. We use the following benchmarks, among others: `chandra-n` [Chandra and Markowsky, 1978] and `mxky` [Kean and Tsiknis, 1990] are problems with known exponential behavior; `adder-n` (n-bit adders), `serial-adder` (n-bit adders obtained by connecting n full adders), `bnp-n-p` (tree-structured circuits with n layers, where p is a probability, [El Fattah and Dechter, 1995]) represent digital circuits; `serial-diag-adder` and `bnp-ab` represent the same circuits in a form suitable for diagnostic reasoning; `pipes` and `pipes-syn` are problems drawn from qualitative physics; `type-k-n` problems,

[1] On the Dimacs [Dimacs, 1992] machine scale benchmark, this Pentium-II 400MHz has a time saving of 305%

Benchmark	#PIs	picomp	zres-tison
adder-10	826457	171	1.07
adder-50	1.0e+25	–	172
adder-100	7.2e+48	–	381
chandra-21	2685	0.1	0.29
chandra-100	9.2e+15	–	12.23
chandra-400	2.8e+63	–	318
bnp-5-50-ts4-rr-2	1240	0.02	0.42
bnp-7-50-ts4-rr-1	5.7e+6	–	10.9
bnp-7-50-ts4-rr-2	3.0e+10	–	15.9
bnp-ab-5-50-ts4-rr-2	1641	0.04	0.80
bnp-ab-7-50-ts4-rr-2	7.2e+10	–	311
m6k6	823585	1.18	0.36
m9k8	1.0e+8	–	1.81
m30k25	1.9e+37	–	256
n-queens-5	3845	1.72	4.26
pigeons-5-5	7710	12.4	49.6
pigeons-6-5	17085	93.5	356
pipes-simple-08	71900	390	38.9
pipes-simple-12	321368	–	239
pipes-syn-10	30848	35.2	2.69
pipes-syn-40	4.3e+6	–	518
serial-adder-10-4	8.4e+7	–	4.33
serial-adder-30-4	6.4a+21	–	717
serial-diag-adder-10-4	1.7e+10	–	78.3
serial-diag2-8-0b	2.2e+8	–	11.7
type1-800	319600	16.4	219
type5-150	3.7e+71	–	330

Table 1: Prime implicates with picomp and zres-tison

due to [Mathieu, 1991], attempt to mimic "typical" structures of expert system KBs, with each k representing a particular kind of structure.

The first observation that can be drawn from Table 1 is the scalability of `zres-tison`, which can handle a huge number of clauses on some particular instances (e.g. 10^{71} for `type5-150`). Thus it forces us to completely reconsider Tison's algorithm efficiency and limitations. In all these examples, `picomp` simply cannot physically represent such sets of clauses and, like all previous Tison's implementations, is no competition for `zres-tison`. Yet, on some instances, `picomp` remains faster. This is due to small distributions computation, involving less than a thousand clauses (e.g. `type-1-800`) at each multiresolution step. On such examples, traditional approaches with explicit representation of clauses remain faster. Notice finally the relative independence of the running time of `zres-tison` w.r.t. the number of PIs, in contrast to `picomp`. For `zres-tison` there are easy examples with a huge number of PIs, and hard examples with few PIs. The latter include `n-queens` and satisfiable `pigeon` problems, which contrasts with the excellent results for satisfiability of similar problems in [Chatalic and Simon, 2000c].

5.2 Polynomial size compilation

One of the major theoretical limits of knowledge compilation (KC) is that it is very unlikely that a KC method can be defined that always produces tractable compiled theories of polynomial size, unless we restrict the query language to have polynomial size. That's what $\mathcal{L}_K$-BE does. To implement this with `zres`, clauses having more than k processed variables are moved to the dead ZBDD. We test here only the `zres-kbound` implementation with different k on the same benchmarks used on prime implicates. Summarizing so many test runs on so many different benchmarks families is not easy. We attempt to do so in Table 2, where a cell on row i and column j contains the percentage of benchmarks quickly solved by the program i in comparison with program j. For instance, `zres-kbound`(3) terminates more quickly than `zres-tison` in 31% of the cases, and the median value of the cpu gain is 3.70. Of course, such a table can't take into account the particularities of each benchmark family, but it still can give some taste of how the different approaches behave with respect to each other.

	k(3)	k(4)	k(5)	k(6)	z-tison
k(3)	–	52 (1.19)	46 (1.45)	46 (1.45)	31 (3.70)
k(4)	31 (1.04)	–	41 (1.18)	41 (1.18)	23 (2.75)
k(5)	37 (1.05)	35 (1.06)	–	0	18 (2.21)
k(6)	37 (1.05)	35 (1.06)	0	–	18 (2.21)
z-tison	53 (2.88)	56 (2.83)	58 (2.88)	58 (2.88)	–

Table 2: Compilation with zres-kbound and zres-tison

What is striking is that Tison remains the fastest procedure in most cases, with a median cpu gain of around 285%. But, we can see that, in 31% of the cases, kbound(3) perform much better than `zres-tison`. This phenomenon (also observed with kbound(4-6)) means that when `zres-tison` fails, kbound remains an available solution. Moreover, our results suggest that kbound(5) and kbound(6) perform similarly, yet the latter computes a much more interesting base.

5.3 Directional resolution

Directional resolution compiles a theory into an equivalent one in which one can generate any model in linear time. We test here our two DR implementations, `drcomp` and `zres`, on some specific benchmarks, with two heuristics for each one. In Table 3 the `par-8-*`, `pret150-25`, `ssa0432` files are taken from the Dimacs database, and #KB means the size of the theory obtained by the programs. As this size depends on the order of variable deletions, we only report it for the first case, where the two programs give the same result.

As shown in Table 3, good heuristics have a major effect. Using just input ordering often forces the size of the KB to grow in such a way that `drcomp` can't compete with `zres`. ZBDDs pay off in all these cases, showing their ability to represent very large KBs. The min-diversity heuristics seems to invert the picture, mainly because the ordering helps to keep the clause set to manageable size (tens of thousands of clauses). The same applies to theories which are tractable for directional resolution, such as circuit encodings [del Val, 2000b], for which `drcomp` takes no time. `zres`, in contrast, seems to include a significant overhead on theories which do not generate many clauses, despite its relatively good performance reported in [Chatalic and Simon, 2000b]. On the other

Bench.	Input Order #KB	zres	drcomp	Min Diversity zres	drcomp
adder-400	7192	482	1.36	310	1.23
chandra-400	1.3e+36	132	–	28.2	0.02
m8k7	2396801	1.19	24.05	0.50	0.01
m30k25	8.76e+37	292	–	79.1	0.05
par8-2-c	15001	332	–	1.24	0.04
par8-2	350	–	476.82	30.31	0.10
pret150-25	1.75e+15	8.69	–	4.83	20.8
ssa0432-003	2.01e+9	407	–	45.2	0.01
type1-850	849	99.9	0.08	91.5	0.11

Table 3: Directional resolution algorithm

hand, ZBDDs scale much better when dealing with very large KBs.

5.4 Abduction

Given a theory Σ and a set A of variables (called assumptions, hypothesis or abducibles), an *abductive explanation* of a clause C wrt. Σ is a conjunction L of literals over A such that $\Sigma \cup L$ is consistent and entails C (or a subsumed clause of C). L is a minimal explanation iff no subset of it is also an explanation.

On the `adder-n`, `mult-n` (adder and multiplier whose result is smaller than n), and on the `bnp` circuits introduced in section 5.1, we are given a circuit Σ with a set of input variables I and output variables O. We want to explain some observation (an instantiation of some O variables) with only variables from I (thus forgetting all internal variables). This problem can be addressed through $\mathcal{L}_V$-BE with $V = I \cup O$. In this case, the trie-based implementation failed in all benchmarks (except `adder-5`), so we only report results for the ZBDD-based implementation, in Table 4. We also tried our algorithms on the ISCAS circuits benchmark family, without success. To interpret these results appropriately, one should note that obtaining $PI_{\mathcal{L}_V}$ means precomputing answers for *all* abduction problems for Σ, as explanations can be directly read off from $PI_{\mathcal{L}_V}(\Sigma)$. Kernel resolution also provides methods for answering only specific abduction problems (as opposed to all), but we have not tested them; for some tests with a similar flavor, check the diagnosis experiments on ISCAS benchmarks, section 5.6.

Note that, surprisingly, all adder examples are treated much more efficiently by `zres-tison` (table 1) than by $\mathcal{L}_V^1$-BE. A similar phenomenon was already pointed out in [Chatalic and Simon, 2000a], where, on some examples, computing DR on a subset of variables was proved harder than on the whole set of variables ($\mathcal{L}_V^0$-BE harder than $\mathcal{L}_A^0$-BE, where $V \subseteq A$). But, here, in addition, the induced hardness overcomes the simplification due to the use of DR instead of Tison, which is clearly counter-intuitive. This is not the case for the `mult-n` nor for the `bnp-10-50` instances, on which `picomp` and `zres-tison` failed.

Table 5 presents some easier examples, the `path-kr` problems from [Prendinger *et al.*, 2000] (abbreviated `p-kr`), which can be solved by both `drcomp` and `picomp`. For comparison with their results, note that we, unlike them, are

Bench.	L_V^0 Time	$\#L_V$-LUB	L_V^1 Time	$\#PI_{L_V}$
adder-5	0.18	448	0.3	1396
adder-10	0.78	18184	3.6	354172
adder-20	3.51	1.88e+7	36.51	2.09e+10
adder-25	5.88	6.03e+8	–	–
adder-100	423.51	2.28e+32	–	–
mult-16	0.66	275	4.47	1918
mult-64	16.86	3466	–	–
mult-128	2.25	1307	585.99	39398
bnp-7-50-1	6.12	193605	6.39	193605
bnp-7-50-3	5.97	24435	6.33	24435
bnp-10-50-3	805.53	2.30e+14	–	–
bnp-10-50-4	671.61	1.76e+19	–	–

Table 4: Abduction on circuits: ZBDD-based L_V-BE.

Bench.	#KB	zres	drcomp	#PIs	zres-tis.	picomp
p-2r	28	0.24	0.03	80	0.30	0.03
p-6r	7596	31.3	0.33	1332	48.5	0.36
p-10r	99100	374	18.7	122000	360	22.9

Table 5: Abduction on path problems

solving all abduction problems for the given instance in one go.

5.5 Fault-Tree Diagnosis

The set of Aralia [Rauzy, 1996] benchmarks arise from fault tree diagnosis, in which one wants to characterize the set of elementary failures (coded by the f_i predicates) that entail the failure of the whole system (the $root$ predicate). It thus amounts to computing the set of L_V*-implicants* of the $root$, where V is the set of f_i. We use the well-known duality cnf/dnf and implicants/implicates to express those problems as L_V*-implicates* problems.

Usually, these benchmarks are efficiently solved using *traditional* BDDs approaches (*i.e.* rewritting the initial formula under Shannon Normal Form). In [Rauzy, 1996], internal variables (defined as an equivalency with a subformula), are not even encoded in the BDD: they are directly identified with the BDD encoding the subformula. In our approach, such internal variable are explicitly present in the cnf of the formula. The work of `zres` amounts to precisely *delete* all of them, thus obtaining a formula build on f_i (and the $root$) predicates. In table 6, we show that kernel resolution, with multiresolution, can efficiently handle those problems without needing to rewrite the formula under Shannon Normal Form. To our knowledge, this is the first time that such direct CF approach successes.

5.6 Model-based diagnosis

In diagnosis [de Kleer *et al.*, 1992], we are given a theory Σ describing the normal behavior of a set of components; for each component, there is an "abnormality" predicate ab_i. Let $V = AB$ be the set of ab_i's. Given a set of observations O, the diagnosis are given by the prime $\mathcal{L}_V$*-implicants* (noted PA_{L_V}) of $PI_{\mathcal{L}_V}(\Sigma \cup O)$. As we can obtain these implicants

Benchmark	zres	$\#PI_{L_V}$
baobab2	1.9	1259
baobab3	16.32	24386
das9205	0.61	17280
das9209	11.98	8.2e+10
edf9205	15.4	21308
edf9206	669.3	7.15e+9
isp9602	17.05	5.2e+7
isp9605	1.67	1454

Table 6: Aralia abduction benchmarks

Bench.	T. L_V^0-BE	$\#L_V$-LUB	T. PA_{L_V}	$\#PA_{L_V}$
c432-d-03	99.16	1648704	315.13	1243
c432-d-06	462.16	3549876	–	–
c432-d-09	140.60	18084	180.67	6651166
c499-d-03	42.14	102	470.12	7347194
c499-d-04	43.39	227	–	–
c499-d-09	68.51	1065	198.74	3.2e+7
c880-d-03	98.16	174	103.13	13349
c880-d-08	98.96	87	99.65	698
c880-d-05	153.35	1	153.47	23
c1355-d-01	268.48	965	–	–
c1355-d-02	233.46	182	–	–
c1355-d-10	224.89	11	225.57	49856

Table 7: Model-based diagnosis for ISCAS benchmarks

from any equivalent L_V-LUB of Σ, either L_V^1-BE or L_V^0-BE can yield the diagnosis, so we use the cheaper L_V^0-BE.

We first consider ISCAS benchmarks. We generated for each of the four easiest circuits their normal behavior formula (by introducing ab_i variables on each gate), and 10 faulty input-output vectors. The results are given in Table 7 (with T. denoting the total time), only for the ZBDD-based implementation. In a number of cases (where L_V-LUB is of reasonable size), `drcomp` can also go through the first phase, but our current implementation fails to complete the second phase, thus we don't report its results here. To generate the implicants with our ZBDD-based system, we use one of its procedures that can negate a given formula. 3 of the 40 instances were too hard for L_V^0-BE, and 14 failed to compute the L_V-implicants. To our knowledge, this is the first time such instances are tested with a direct consequence-finding approach.

Table 8 presents results for `bnp-ab` circuits for the much harder "compiled" approach to diagnosis, where we basically precompute $PI_{\mathcal{L}_V}$ for all possible input-output vectors, by computing $PI_{\mathcal{L}_{AB\cup O\cup I}}$. Clearly, in this case it pays off to use the vocabulary-based procedures (as opposed to full PIs) for both tries and ZBDDs.

5.7 Random instances

Finally, we consider random instances generated with the fixed clause-length model. While conclusions drawn from them do not say anything about the behavior of algorithms on structured and specially real-world instances, they provide in our case a good tool for evaluating subsumption and optimization techniques for resolution. It was observed in [Dechter and Rish, 1994; Chatalic and Simon, 2000a] that such instances are very hard for DR, and [Schrag and Crawford, 1996] observed even worse behavior for consequence-finding for instances generated near the SAT phase transition threshold. Using a more powerful computer than in the previous benchmarks[2], we compare `picomp`, `zres-tison`, and `zres-kbound` on 1000 formulae at the ratio 4 (30 variables, 120 clauses, 80% satisfiable). If we focus on satisfiable theories, their mean number of prime implicates is 494 (std: 1595; median: 65; 50% confidence interval: 30-254; max: 20000). Table 9 summarizes cpu-time results (the last column is the number of unsolved instances and the maximal cpu time needed if all instances have been solved).

Bench.	#KB	zres	drcomp	#PIs	zres-tis.	picomp
bnp-5-1	200	0.54	0.03	6441	0.90	0.24
bnp-5-3	35	0.48	0.03	724	1.02	0.03
bnp-7-1	78109	19.26	21.6	1.3e+8	30.4	–
bnp-7-2	31166	8.55	52.1	1.7e+11	33.9	–
bnp-7-4	110829	13.89	–	8.2e+8	32.8	–

Table 8: "Compiled" diagnosis on tree-structured circuits

Program	Mean	Med.	Std	50% int.	#Uns.(MT)
zres-tison	8.15	3.43	14.03	1.32-8.95	0 (232)
picomp	32.79	9.79	64.61	2.72-27.78	42
zres-kbound 3	3.78	2.67	3.61	1.57-4.71	0 (34.3)
zres-kbound 4	7.48	3.93	10.28	1.83-8.66	0 (152)
zres-kbound 5	8.86	3.97	14.52	1.81-9.61	0 (218)

Table 9: Consequence Finding in Random Formulae

We observe from this table that ZBDDs are always better than tries on random instances. For instance, with `zres-tison`, the maximal number of clauses reached by each run has a median of 4890 (mean; 5913, std: 4239, 50% int: 2891-8022, max: 30147), but the maximal number of ZBDDs nodes has only a median of 3393 (mean: 3918, std: 2522, 50% int: 2096-5236, max: 19231). This gives less than one node per clause on average, despite the fact that most of the clauses are longer than 4. Even if such instances are "unstructured", at the beginning of the calculus, resolution (which we have seen really amounts to clause distribution) introduces many redundancies in formulae, which seem to be well-captured by ZBDDs.

6 Conclusion

We presented an extensive experimental study of consequence-finding, including results on many of its applications, using two quite different implementations. By combining the focusing power of kernel resolution with the ZBDD representation and multiresolution, we have shown the scalability of our approach and the relative independance of its performances with respect to the size of the handled KB. The until now mainly theoretical applications of CF can now be approached for problems of realistic size, and in some cases of spectacular size.

[2] AMD-Athlon 1GHz running Linux with a Dimacs machine scale of 990% cpu savings.

References

[Bryant, 1992] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[Cadoli and Donini, 1997] M. Cadoli and F. M. Donini. A survey on knowledge compilation. *AI Communications*, 10:137–150, 1997.

[Chandra and Markowsky, 1978] A. K. Chandra and G. Markowsky. On the number of prime implicants. *Discrete Mathematics*, 24:7–11, 1978.

[Chatalic and Simon, 2000a] Ph. Chatalic and L. Simon. Davis and Putnam 40 years later: a first experimentation. Technical report, LRI, Orsay, France, 2000.

[Chatalic and Simon, 2000b] Ph. Chatalic and L. Simon. Multi-resolution on compressed sets of clauses. In *Proc. ICTAI (Tools with AI)*, pp. 449–454, 2000.

[Chatalic and Simon, 2000c] Ph. Chatalic and L. Simon. Zres: The old Davis–Putnam procedure meets ZBDD. In *CADE'00, Int. Conf. Automated Deduction*, pp. 2–10, 2000.

[Davis and Putnam, 1960] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[de Kleer *et al.*, 1992] J. de Kleer, A. K. Mackworth, and R. Reiter. Characterizing diagnosis and systems. *Artificial Intelligence*, 56:197–222, 1992.

[de Kleer, 1986] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[de Kleer, 1992] J. de Kleer. An improved incremental algorithm for generating prime implicates. In *AAAI'92, Proc. 10th Nat. Conf. on Artificial Intelligence*, pp. 780–785, 1992.

[Dechter and Rish, 1994] R. Dechter and I. Rish. Directional resolution: The Davis-Putnam procedure, revisited. In *KR'94, Proc. 4th Int. Conf. on Knowledge Representation and Reasoning*, pp. 134–145. Morgan Kaufmann, 1994.

[del Val, 1995] A. del Val. An analysis of approximate knowledge compilation. In *IJCAI'95, Proc. 14th Int. Joint Conf. on Artificial Intelligence*, pp. 830–836, 1995.

[del Val, 1999] A. del Val. A new method for consequence finding and compilation in restricted languages. In *AAAI'99, 16th Nat. Conf. on Artificial Intelligence*, pp. 259–264, 1999.

[del Val, 2000a] A. del Val. The complexity of restricted consequence finding and abduction. In *AAAI'2000, Proc. 17th Nat. Conf. on Artificial Intelligence*, pp. 337–342, 2000.

[del Val, 2000b] A. del Val. Tractable classes for directional resolution. In *AAAI'2000, Proc. 17th Nat. Conf. on Artificial Intelligence*, pp. 343–348, 2000.

[Dimacs, 1992] The Dimacs challenge benchmarks. Originally available at ftp://ftp.rutgers.dimacs.edu/challenges/sat.

[El Fattah and Dechter, 1995] Y. El Fattah and R. Dechter. Diagnosing tree-decomposable circuits. In *IJCAI'95, Proc. 14th Int. Joint Conf. on Artificial Intelligence*, pp. 1742–1748, 1995.

[Inoue, 1992] K. Inoue. Linear resolution for consequence-finding. *Artificial Intelligence*, 56:301–353, 1992.

[Kean and Tsiknis, 1990] A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9:185–206, 1990.

[Kean and Tsiknis, 1993] A. Kean and G. Tsiknis. Clause management systems. *Computational Intelligence*, 9:11–40, 1993.

[Marquis, 1999] P. Marquis. Consequence-finding algorithms. In D. Gabbay and Ph. Smets, editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems (vol. 5): Algorithms for Defeasible and Uncertain Reasoning*, pp. 41–145. Kluwer Academic Publishers, 1999.

[Mathieu, 1991] Ph. Mathieu. *L'utilisation de la logique trivaluée dans les systèmes experts*. PhD thesis, Université des Sciences et Technologies de Lille, France, 1991.

[Minato, 1993] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 272–277, 1993.

[Prendinger *et al.*, 2000] H. Prendinger, M. Ishizuka, and T. Yamamoto. The Hyper system: Knowledge reformation for efficient first-order hypothetical reasoning. In *PRICAI'00, 6th Pacific Rim Int. Conf. on Artificial Intelligence*, 2000.

[Rauzy, 1996] A. Rauzy. Boolean models for reliability analysis: a benchmark. Tech. report, LaBRI, Univ. Bordeaux I.

[Reiter and de Kleer, 1987] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems. In *AAAI'87, 6th Nat. Conf. on Artificial Intelligence*, 1987.

[Schrag and Crawford, 1996] R. Schrag and J. Crawford. Implicates and prime implicates in random 3–SAT. *Artificial Intelligence*, 81:199–221, 1996.

[Selman and Kautz, 1996] B. Selman and H. Kautz. Knowledge compilation and theory approximation. *Journal of the ACM*, 43(2):193–224, 1996.

[Selman and Levesque, 1996] B. Selman and H. J. Levesque. Support set selection for abductive and default reasoning. *Artificial Intelligence*, 82:259–272, 1996.

[Tison, 1967] P. Tison. Generalized consensus theory and application to the minimization of boolean circuits. *IEEE Transactions on Computers*, EC-16:446–456, 1967.