Automatic Generation of Implied Clauses for SAT

Lyndon Drake, Alan Frisch, and Toby Walsh

Artificial Intelligence Group, Department of Computer Science, University of York, York YO10 5DD, United Kingdom,
{lyndon,frisch,tw}@cs.york.ac.uk,
http://www.cs.york.ac.uk/~{lyndon,frisch,tw}

Abstract. This paper presents a survey of the use of resolution in propositional satisfiability, and some preliminary results from the implementation of a hybrid resolution and search algorithm for SAT.

1 Introduction

Davis-Putnam (DP) was the first practical complete algorithm for solving propositional satisfiability (SAT) problems [5]. DP uses resolution to determine whether a SAT problem instance is satisfiable. However, resolution is not always practical, as it can use exponential space and time. The most important refinement to DP was DLL [4], which replaced the resolution in DP with splitting (backtracking search). Backtracking search still uses exponential time for SAT in the worst case, but only needs linear space. As time is more readily available than space, the change to search was a big improvement.

Since then, backtracking search has been used almost exclusively for complete SAT solvers [7]. In this paper we present some preliminary results of an investigation into the use of resolution in a SAT solver. These results are in the context of complete SAT algorithms, rather than local search algorithms. Complete algorithms always determine whether a problem instance is satisfiable. Using resolution in conjunction with search can perform better than pure search on some problems. In particular, structured problems are more likely to benefit from adding resolution to search, as it is more likely that useful inferences will be available in a structured problem.

2 Propositional satisfiability problem

The propositional satisfiability problem was the first NP-complete problem to be identified [3]. Any other NP-complete problem is reducible to SAT in polynomial time, which means among other things that improvements in the performance of SAT algorithms are useful in a number of other fields.

Some reductions into SAT that are competitive with domain-specific solvers include planning [10, 11], symbolic model checking [1], quasigroup completion problems, and hardware modelling [6].

SAT is defined as follows: given a propositional formula, is there an assignment of truth values to the propositions such that the formula is satisfied? SAT problems are

typically given in conjunctive normal form. For example, given the satisfiable instance in Table 1, a satisfying assignment is $a = true$, $b = false$, $c = false$.

Obviously, not all SAT problems are satisfiable. For example, the second formula in Table 1 is unsatisfiable, because there is no possible assignment of values to variables that satisfies every clause in the formula. Note that this is an example of a 3-SAT problem, as there are exactly three variables per clause. In general, k-SAT problems involve k variables per clause.

Satisfiable instance	**Unsatisfiable instance**
$a \vee b \vee \neg c$	$a \vee b \vee c$
$\neg a \vee \neg b$	$a \vee b \vee \neg c$
$b \vee \neg c$	$a \vee \neg b \vee c$
	$a \vee \neg b \vee \neg c$
	$\neg a \vee b \vee c$
	$\neg a \vee b \vee \neg c$
	$\neg a \vee \neg b \vee c$
	$\neg a \vee \neg b \vee \neg c$

Table 1. Two propositional formulae

2.1 Algorithms for SAT

DP and DLL are very similar procedures (see Figures 1 and 2). The difference between them is that DP uses resolution while DLL uses splitting. Splitting is assigning a value to a proposition, and then calling DLL on the formula as modified by the assignment.

For example, in the unsatisfiable formula given in Table 1, making the assignment $a = true$ would satisfy the first four clauses. Those clauses can then be removed from the formula, the $\neg a$'s removed from the remaining four clauses, and the modified formula $(b \vee c) \wedge (b \vee \neg c) \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$ passed to DLL for a further step in the search process.

procedure DP(Σ)
  if $\{\} \in \Sigma$ **then** return $false$
  if $\Sigma = \{\}$ **then** return $true$
  if $\{l\} \in \Sigma$ **then** return DP($\Sigma[l/true]$)
  $l :=$ choose a literal in Σ
  return DP($(\Sigma \wedge l) \vee (\Sigma \wedge \neg l)$)

Fig. 1. The DP algorithm for propositional satisfiability.

The most common extension to DLL is the addition of a branching heuristic. Branching heuristics select the next variable to split on, and also determine whether a true or

procedure DLL(Σ)
  if $\{\} \in \Sigma$ **then** return $false$
  if $\Sigma = \{\}$ **then** return $true$
  if $\{l\} \in \Sigma$ **then** return DLL($\Sigma[l/true]$)
  $l :=$ choose a literal in Σ
  $v :=$ choose a value for l in $\{true, false\}$
    if DLL($\Sigma[l/v]$) **then** return $true$
    else return DLL($\Sigma[l/\neg v]$)

Fig. 2. The DLL algorithm for propositional satisfiability.

false assignment should be tried first. In Figure 2, the branching heuristic is the statements:

$l :=$ choose a literal in Σ
$v :=$ choose a value for l in $\{true, false\}$

The choice of branching heuristic is an extremely important part of a DP implementation [8], because a correct decision at each node in the search tree will result in a backtrack-free search. However, as SAT is in NP, there will be an exponential worst case problem instance for any heuristic (assuming that $\mathrm{P} \neq \mathrm{NP}$).

2.2 Resolution in SAT solvers

Unit resolution is used in every complete SAT solver, as it is extremely effective. Typically many more unit resolutions than splits (variable assignments) are used in testing the satisfiability of a problem instance. Unit resolution is included in both DP and DLL (the statement **if** $\{l\} \in \Sigma$ **then** return p($\Sigma[l/true]$)).

Cha and Iwama [2] describe the use of resolution in local search. Whenever their local search procedure (ANC) reaches a local minima, the procedure searches resolves on *neighbouring* clauses. The resolution algorithm is defined in Figure 3. Two clauses are neighbouring iff they differ by the sign of exactly one literal (for example, $a \vee \neg b \vee c$ and $a \vee \neg b \vee \neg c$ are neighbours by this definition, while $a \vee \neg b \vee c$ and $a \vee b \vee d$ are not). After adding the resolvent clauses ($a \vee \neg b$ for this example), the local search process is restarted, with the intention being that the added clauses will guide the search on a different path that avoids getting stuck in the same local minima as before. Their empirical analysis suggests that this method of adding clauses is significantly faster than a similar method that simply increases the weights of the unsatisfied clauses at the minima.

Rish and Dechter [15] compare the use of ordered resolution and backtracking search for complete SAT solving on a number of different SAT problems. They also describe a heuristic, based on the induced width of the problem, which they used to decide when to use resolution. They present both theoretical and empirical results comparing the search and resolution, and suggest several hybrid algorithms which demonstrate the value of combining resolution and complete search. They also confirmed that resolution is more useful on structured problems than on random problems.

```
procedure ANC(Σ)
  U := all currently unsatisfied clauses in Σ at the local minima
  for each clause u ∈ U
    for each clause n ∈ Σ s.t. n differs from u by the sign of exactly one literal
      r := the resolvent of n and u
      Σ := Σ ∪ r
```

Fig. 3. The ANC resolution algorithm

3 Neighbourhood resolution in complete search

We have implemented a hybrid resolution and backtracking search procedure, similar to Cha's ANC local search procedure, in order to investigate the use of neighbourhood resolution in a complete SAT procedure. At each node in the search tree, our solver determines whether any neighbouring clauses exist given the current assignment. If neighbours do exist, it uses a heuristic do decide whether to resolve (currently we use a random heuristic that randomly returns true 10% of the time).

The solver is written in C, and was designed to be easy to modify so that additional features, such as neighbourhood resolution, can be added. Some optimisation has been carried out by profiling the solver on a real problem, determining from the profile which functions consume most of the running time, and modifying the function in question by precalculating as much as possible (either at the start of the search or during assignment).

4 Preliminary results

These results are preliminary and not comprehensive, so there is no guarantee that they are representative. Our SAT solver is not fully optimised, and so the metrics are not comparable to existing solvers. In particular, slow run times have limited the choice of problems that we have been able to test so far.

4.1 (k, m)-tree embeddings

To generate structured problems, we used a (k, m)-tree embeddings problem generator (kindly supplied by Irina Rish and described in [15]). This generates instances by producing SAT mappings of randomly generated clique-chain structures. The generated instances are intended to have properties common to real-life problem instances, and have the advantage of being able to control the generation parameters. This allows us to generate structured instances that can still be solved in a reasonable amount of time.

On trivial instances resolution provides no advantage. However, on some more difficult problem instances, the neighbourhood resolution shows a clear improvement over pure search. For example, on one instance defined by $k = 8$, $m = 12$, 50 cliques per tree, 34 clauses per clique, 3 literals per clause, and a positive literal probability of 0.5, the two algorithms produced the metrics shown in Table 2.

	Search	Neighbourhood resolution
Time	9.39 seconds	6.32 seconds
Nodes	108 214	1 779
Unit propagations	92 591	1 432
Backtracks	7 742	53
Resolutions	n/a	24

Table 2. Performance metrics for (k, m)-tree embeddings problem instance

The main reason why the decrease in time is not proportional to the reduced number of search nodes is that the resolution code is unoptimised. However, the decreases in the number of search nodes and backtracks are very promising.

4.2 SATLIB problems

We have started testing the neighbourhood resolution code on some SATLIB [9] problem instances.

On instances 6 and 7 of the DIMACS pigeonhole problems, there is no difference between the pure search and resolution code, as the resolution code does not find any resolutions to perform. Similarly, on the DIMACS parity checking problems and SATLIB uniform random problems that we have tested so far, the resolution code fails to find any resolutions to perform.

These results suggest that either the random heuristic used to decide when neighbourhood resolution is appropriate needs to be changed, or the definition of neighbourhood needs to be relaxed for some problem classes.

5 Future work

Our aim is to extend this work in the following ways:

1. General improvements to the SAT solver code. The main aim of the improvements will be to increase the performance of the code, in order to make running time comparisons with other SAT solvers possible. Particular areas of the solver that can be improved are the calculation of heuristics, the addition of a value ordering heuristic, and the addition of the pure literal rule.
2. Improvements to the neighbourhood resolution code. Currently, the neighbourhood resolution is implemented very simply. There are a number of possible optimisations that could be applied, especially to the search for neighbouring clauses. In general, we want to apply the same principle to optimising the resolution code as we did to optimising the backtracking code: that is, by precomputing as much as possible, either before starting the procedure or during assignment.
   We also intend to explore better heuristics for deciding when to perform resolution. Currently, a random heuristic is used so that resolution is carried out about once every ten search nodes (if a resolution is possible at that node).

Finally, a heuristic is needed to choose between several possible resolutions when more than one resolution is possible. For example, resolutions that generate short clauses (e.g. unit clauses) should be preferred over other resolutions.

3. Identify favourable structural features. We expect that some problems will prove to be more suited to the use of inference because of structural features of the problem class. By carrying out testing to identify problem instances for which inference improves performance, and analysis of the problem classes to identify the relevant features, we hope to be able to classify other problem instances using those features in order to make a decision on whether inference will be worthwhile.
4. Investigate other forms of resolution and inference. For example, conflict analysis (or nogood learning) is a very useful technique for many problems [12, 14]. It is used in both GRASP [13] and SATO [16], which appear to be the fastest publicly available SAT solvers. Conflict analysis is a form of inference, and we want to compare conflict analysis and resolution to see if they are useful on the same problems, and if so whether they are orthogonal to each other.

6 Conclusions

Using resolution in conjunction with search seems to be show some promise. On certain problems, resolution can provide dramatic improvements in performance over pure backtracking search, in terms of both the number of search nodes explored and runtime used. Further work needs to be carried out to determine whether the performance increases observed are representative of the problem types, and to extend the work to compare neighbourhood resolution with other similar methods.

References

1. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., July 1999.
2. Byungki Cha and Kazuo Iwama. Adding new clauses for faster local search. *Proceedings of AAAI-96*, 1996.
3. S.A. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third ACM Symposium on Theory of Computing*, pages 151–158, 1971.
4. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
5. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
6. Luis Guerra e Silva, Joao P. Marques Silva, Luis M. Silveira, and Karem A. Sakallah. Timing analysis using propositional satisfiability. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, September 1998.
7. Jun Gu, Paul W. Purdon, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (sat) problem: A survey. In *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 19–152. American Mathematical Society, 1997.

8. John N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.
9. Holger H. Hoos and Thomas Stützle. *SATLIB: An Online Resource for Research on SAT*, pages 283–292. Frontiers in Artificial Intelligence and Applications. IOS Press, 2000.
10. Henry Kautz and Bart Selman. Planning as satisfiability. In J. LLoyd, editor, *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 359–379, 1992.
11. Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 318–325, 1999.
12. Joao P. Marques Silva and Karem A. Sakallah. Conflict analysis in search algorithms for satisfiability. *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, Nov 1996.
13. Joao P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5), may 1999.
14. E Thomas Richards and Barry Richards. *Non-systematic Search and No-good Learning*, pages 107–151. Frontiers in Artificial Intelligence and Applications. IOS Press, 2000.
15. Irina Rish and Rina Dechter. *Resolution versus Search: Two Strategies for SAT*, pages 215–259. Frontiers in Artificial Intelligence and Applications. IOS Press, 2000.
16. Hantao Zhang and Mark E. Stickel. *Implementing the Davis-Putman Method*, pages 309–326. Frontiers in Artificial Intelligence and Applications. IOS Press, 2000.