The Interaction Between Inference and Branching Heuristics

Lyndon Drake and Alan Frisch

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

Abstract. We present a preprocessing algorithm for SAT, based on the HypBinRes inference rule, and show that it does not improve the performance of a DPLL-based SAT solver with conflict recording. We also present evidence that the ineffectiveness of the preprocessing algorithm is the result of interaction with the branching heuristic used by the solver.

1 Introduction

Propositional satisfiability (SAT) is the archetypal NP-complete problem [3]. It is possible to solve a wide range of problems from classes such as constraint satisfaction [18], quasigroup completion [19], and model checking [2], by encoding them into SAT and solving the SAT representation of the problem. In this paper we assume that SAT problem instances are given in conjunctive normal form (CNF), where a formula is the conjunction of a set of clauses. Each clause consists of a disjunction of literals, where a literal is either an atom or its negation.

The fastest available complete SAT solvers (such as Chaff [15]) use variants of the DPLL [5, 4] search algorithm for SAT. These solvers typically augment the search procedure with two fast inference techniques: unit propagation (included in the original DPLL algorithm) and conflict clause recording [14].

In this paper we present a preprocessor for complete SAT solvers, based on a binary clause inference rule called HypBinRes [1]. We show that this preprocessor does not improve the performance of Chaff, and give evidence that this is due to interaction with the branching heuristic used by Chaff.

2 HypBinRes Preprocessing

HypBinRes (or hyper-binary resolution) is a rule for inferring binary clauses. These binary clauses are resolvents of the original formula, but applying the HypBinRes rule can discover these binary consequences more efficiently than resolution. We first define the HypBinRes rule and discuss its implementation as a preprocessor for a SAT solver, and then give experimental results that demonstrate that HypBinRes preprocessing is ineffective at improving the performance of Chaff on a wide range of SAT instances.

2.1 The HypBinRes Rule

Given the literals $a_1, \ldots, a_n$ and h, where $\overline{a}_i$ is the complement of a_i, an example of applying HypBinRes is:

$$\frac{\begin{array}{c} a_1 \vee a_2 \vee a_3 \\ \overline{a}_1 \vee h \\ \overline{a}_2 \vee h \end{array}}{a_3 \vee h}$$

More formally, given a clause of the form $(a_1 \vee \cdots \vee a_n)$ and a set of $n-1$ satellite clauses of the form $(\overline{a}_i \vee h)$ each containing a distinct $a_i \in \{a_1, \ldots, a_n\}$, we can infer the clause $(a_x \vee h)$ where $a_x \in \{a_1, \ldots, a_n\}$ and $a_x \neq a_i$ for all $i \neq x$. This inference rule, called HypBinRes, corresponds to a series of binary resolution [16] steps resulting in a binary clause, and was defined by Bacchus [1]. It is more efficient to use HypBinRes to infer a set of binary clauses than to use resolution to find the same set of clauses, because resolution may take several steps and generate intermediate resolvent clauses.

De Kleer [6] describes an almost identical resolution rule called H5-3 which also infers binary clauses. The H5-3 rule differs by restricting operation to inference on negative satellite clauses, and by not requiring binary satellite clauses in the resolution.

2CLS+EQ [1] is a complete DPLL-based SAT solver that uses HypBinRes during search. 2CLS+EQ is unusual in its use of complex reasoning during search, and is the only publically available solver that uses non-unit resolution during search to achieve comparable performance to the fastest conventional SAT solvers.[1]

2.2 Using HypBinRes in a Preprocessor

While 2CLS+EQ is competitive, it is still outperformed by the best conventional SAT solvers. We decided to examine the effect of HypBinRes as a preprocessor for such a conventional SAT solver. Adding HypBinRes-implied clauses to SAT instances could result in the search space being pruned, and as a result improve runtime performance.

The effect of preprocessing using HypBinRes may vary from problem class to problem class. The constraint satisfaction problem (CSP) is another NP-complete problem, for which two main SAT encodings exist: the direct encoding [18], and the support encoding [10]. The support encoding of a CSP instance generally results in better SAT solver runtime performance than the direct encoding, as DPLL on the support encoding enforces arc consistency [11] on the formula. Furthermore, we have previously shown [8] that HypBinRes, when applied to the direct encoding will generate support clauses for the functional

[1] Van Gelder's 2clsVER [17] uses other reasoning during search, but its performance is not competitive with the fastest solvers.

constraints in the instance. As a consequence, we might expect improved performance on SAT-encoded CSP instances after preprocessing with HypBinRes.

Experiments with 2CLS+EQ showed that for many other problem classes, a reasonable number of HypBinRes clauses can be generated during preprocessing. In order to make 2CLS+EQ perform well, its data structures and heuristics have been designed to interact efficiently with HypBinRes resolution during search. As a result, it is not a particularly efficient solver when used with HypBinRes disabled during search, but the number of inferences made during preprocessing suggested to us the hypothesis that a separate HypBinRes preprocessor would improve the performance of a more conventional SAT solver.

2.3 Results

In order to test this hypothesis, we wrote a preprocessor (based on Swan, our SAT solver) to apply HypBinRes. The preprocessor takes as input a file in DIMACS CNF format [7], and produces as output a CNF file containing the HypBinRes resolvents of the formula in addition to the original clauses. Our implementation of HypBinRes is not the most efficient possible, but the preprocessor performs fast enough that the runtime required by the preprocessor is neglible compared to the solver runtime on most instances.

Fig. 1 shows the results of comparing Chaff runtime performance at solving 100 hard random CSP instances using the direct encoding,[2] before and after applying HypBinRes as a preprocessor. The instances included a mix of satisfiable and unsatisfiable problems. Points below the line indicate improved performance resulting from the preprocessing, while those above the line show decreased performance. All times are given in seconds, and all the experiments in this section were carried out on a Pentium III 650 machine with 192MB of RAM.

The overall impact of using HypBinRes as a preprocessor for Chaff is negligible: the geometric mean runtime for the original problem instances is 1.74 seconds, while the corresponding mean for the preprocessed instances is 1.73 seconds (the medians were identical). However, performance on individual problem instances varies greatly, with some problems displaying much better and others much worse performance (with an interquartile range of 2.9s for the original instances and 3.1s for the preprocessed instances).

These results for HypBinRes preprocessing were not confined to SAT-encoded CSP instances, and in fact all problem classes in the SATLIB library [12] display substantially similar behaviour (though there are minor variations). For example, Fig. 2 shows Chaff's performance on 22 quasigroup (Latin squares) problems. The quasigroup problems contain a mix of satisfiable and unsatisfiable instances.

[2] While these CSP instances are random, the SAT instances derived using the direct encoding contain some structure as a side-effect of the encoding. That structure can be exploited by HypBinRes [8]. In our experiments, Chaff (noted for its high performance on structured SAT instances) often outperformed SATZ (which performs best on random SAT instances) by one or two orders of magnitude on the SAT-encoded CSP instances.

Fig. 1. Chaff runtimes for CSP instances before and after HypBinRes preprocessing (log-log scale, times in seconds)

Fig. 2. Chaff runtimes for SATLIB QG instances before and after HypBinRes preprocessing (log-log scale, times in seconds)

In all the instances we examined, the change in runtime was matched by an approximately proportional change in the number of search nodes explored. This suggests that the runtimes were not significantly affected by the increased size of the preprocessed formulae, but were instead determined by the size of the search space explored.

2.4 Analysis

Intuitively, while making the inferences required to generate implied clauses must take time and the extra clauses will add some overhead to the search procedure, we would expect the additional clauses to result in a smaller search space. Instead, what we observed was that Chaff sometimes explored a larger search space on a preprocessed formula than on the corresponding original formula. Two possible explanations for this observed behaviour are that the additional clauses interact poorly with the conflict clause generation, and that they interact poorly with the heuristic. Interaction with the conflict clauses is difficult to study, but interaction with the heuristic is more practical to investigate so we examined that first (see Section 3.4 for some results on conflict analysis).

The heuristic used by Chaff is, like many SAT heuristics, based on counting literals. Adding implied clauses is likely to change the relative literal counts by increasing the occurrences of some literals relative to others. If the original heuristic ordering was nearly optimal, this change in relative literal counts may result in degraded performance on the preprocessed formula.

Fig. 3. Density plot of literals in `qg3-09`. The grey line shows the original formula, and the black line the preprocessed formula. Chaff performed worse on the preprocessed formula

Fig. 4. Density plot of literals in `qg5-13`. The grey line shows the original formula, and the black line the preprocessed formula. Chaff performed much better on the preprocessed formula

Two quasigroup instances which demonstrate the variability in performance are `qg3-09`, which took 201 seconds to solve before preprocessing and 212 seconds afterwards, and `qg5-13`, which took 149 seconds to solve before preprocessing and 88 seconds afterwards. If these instances were satisfiable the faster times might simply be explained by the branching heuristic making a fortunate early choice in order to find a single satisfying assignment. However, since both these instances are unsatisfiable, if the branching heuristic is the reason for the the faster times then

If these instances were satisfiable the faster times might simply be explained by the branching heuristic making a fortunate early choice in order to find a satisfying assignment. However, both instances are unsatisfiable, so if the branching heuristic is the reason for the performance differences then it must make .

Figures 3 and 4 show density plots of the literal counts for these two instances. There are many hundreds of literals in these instances – too many for a bar plot of the literal counts. A density plot shows a smoothed version of the counts, while still displaying interesting features. Differences between the two lines in each plot correspond to differences in the literal count between the original and preprocessed formulae. In both cases it is clear that in each instance the relative literal counts have been altered by the additional clauses, and this effect is proportionally much greater in the case of `qg3-09` (which Chaff performed worse on after preprocessing).

The differences between the two plots suggest that poor performance may be the result of the heuristic making poor decisions on the preprocessed formula, due to altered literal counts. It is not possible to be confident on this point based purely on counting literals in the formula, as the actual heuristic decisions in Chaff depend on the conflict clauses generated during search. However, the differences between these two instances and others we studied were enough to motivate further investigation of the interaction between implied clauses and the heuristic.

3 Isolating the Heuristic

One method for investigating the effect of implied clauses on a heuristic is to prevent the heuristic considering the additional clauses. In other words, if we ensure that only the clauses in the original formula are visible to the heuristic, the additional clauses will not significantly change the decision order. The additional clauses might directly terminate a branch early, or result in new unit propagations, but if a static heuristic is used this will not result in a larger search tree. There is potentially a small negative impact on runtime performance from handling a larger formula.

To achieve this in the context of Chaff, we would have to not only hide the additional clauses from the heuristic, but prevent them from being involved in conflict clause generation (as the heuristic used in Chaff is deliberately biased towards literals appearing in recent conflict clauses). We decided instead to implement implied clause hiding in our own SAT solver Swan (which is easy to modify and does not use conflict recording), by annotating the preprocessed formula with hints identifying the additional implied clauses, and altering the heuristic to stop it counting literals in implied clauses.

3.1 Hint syntax

In order to separate the marking of implied clauses from the particular preprocessor or solver implementation being used, we decided to extend the DIMACS CNF format [7]. We did this by giving implied clause "hints" as a comment line in the preamble, of the form `c !! hint implied-clause 2 45 47 0`, where 2, 45 and 47 are clause numbers. Clauses are assumed to be numbered in the order they appear in the formula, starting at 1. The format is intended to be easy for people to read and modify, and for software to parse. For example, Fig. 5 shows a CNF file in which the clauses $(3 \vee 10)$ and $(3 \vee 4)$ have been marked as implied clauses.

Because hints are placed in a comment line, they will be ignored by solvers that have not been modified to interpret the hints. As a consequence of this, a preprocessor that adds implied clauses can include a hint line in its output, and the resulting file still conforms to the DIMACS CNF format specification. Existing SAT solvers, such as Chaff, can be run on the preprocessed formula without modification (the implied clauses will still be used by the solver), while only a relatively simple parsing change is required in order to use the hints.

```
c !! hint implied-clause 5 7 0
p cnf 7 10
1 2 3 4 0
-2 10 0
-1 10 0
-4 10 0
3 10 0
4 -10 0
3 4 0
```

Fig. 5. Example of a CNF file with an implied clause hint

3.2 Results

We tested implied clause hiding on the same set of quasigroup instances as before, using our modified version of Swan, and two static heuristics (one a static version of the VSIDS heuristic used in Chaff). The two heuristics behaved similarly with respect to the visibility of the implied clauses (though different problem classes performed better with one or the other heuristic).

All times are given in seconds, and all the experiments in this section were carried out on a dual processor Pentium III 750 machine with 1GB of RAM. A timeout of 5 minutes was used for the single solution counting experiments (Figures 6 and 7), and a timeout of 20 minutes when we configured Swan to count all solutions of the satisfiable instances. Swan failed to solve a number of the instances within the time limit, though in some cases it solved the preprocessed version of those instances.

There are two main points to note from these results: on average, preprocessing a formula with HypBinRes reduced the runtime by more than 50%; and using hints to hide the additional clauses from the heuristic resulted in no performance improvement after preprocessing. We obtained similar results when we configured Swan to count all solutions on the satisfiable instances. When implied clauses were hidden from the heuristic, the search space on the preprocessed problem was no larger than that on the original problem, as expected (though there was occasionally a small runtime penalty). We also ran experiments on the SAT-encoded CSP instances, but Swan was unable to solve many of the instances within the time limit. However, on the instances that Swan was able to solve we observed similar behaviour to the results for the quasigroup instances.

3.3 Analysis

HypBinRes generates only binary clauses, which are most likely to contribute to a proof if they are involved in unit propagation. When the implied binary clauses are visible, the heuristic is more likely to make decisions resulting in unit propagation on those clauses, as the literals in the implied clauses will receive higher weightings.

Fig. 6. Swan runtimes for SATLIB QG instances before and after preprocessing (log-log scale, times in seconds)

Fig. 7. Swan runtimes for SATLIB QG instances before and after preprocessing, hiding the implied clauses (log-log scale, times in seconds)

The heuristic used by Satz [13] is based on the idea of choosing to assign the literal which will result in the most unit propagations. Bacchus points out that while this heuristic is expensive to compute, given a literal l counting the number of binary clauses containing the negation of l gives the number of unit propagations resulting from assigning l true [1, Observation 6]. The effect on a literal counting heuristic (like those used in Chaff and Swan) of adding HypBinRes clauses will be to give higher priority to assigning those variables which will result in unit propagations (though the value ordering for a particular variable will not be optimal).

These results go some way towards explaining the performance results from Chaff. The heuristic used by Chaff gives precedence to literals appearing in recently generated conflict clauses, so as the search space is explored literals from clauses in the input formula (including those added by HypBinRes) will contribute relatively little to the literal counts. If the search proceeds for long enough, the clauses in the input formula will be effectively ignored in preference to those generated by conflict recording.

However, this explanation cannot completely account for the observed behaviour of Chaff, as on some of the SAT-encoded CSP instance the size of the search space explored by Chaff increased. These larger search spaces must be due either to poor branching heuristic choices early in the search space because of the altered literal counts, or to the generation of different conflict clauses because of the different decisions and unit propagations.

Fig. 8. Search space pruning by implied clauses (denotes a failure node, and nodes which only fail when implied clauses are added to the original formula)

3.4 Further results

In order to determine how much effect the implied clauses were having on the heuristic, we made two further modifications to Swan: first adding a form of conflict analysis, and then quantifying the size of branches pruned by implied clauses.

The conflict analysis uses the implication graph to identify the clauses involved in a conflict, operating in much the same way as the conflict analysis engine in a solver with conflict clause recording. Two differences are that instead of stopping at the first unique implication point (UIP) as Chaff and other similar solvers do, we trace all the way back to the decision variables, and we do not generate a conflict clause. If an implied clause is involved in a conflict, either directly by the current assignment making the clause false, or indirectly by being involved in the sequence of unit propagations that leads to another clause becoming false, then that clause has pruned a branch.

Worth considering is that it might not be necessary to do the full CR in JQuest thing. If the implied clauses are rarely (or never) involved in the conflicts, then they will have no impact on the generated CR clauses. So the experiments to show that implied clauses don't get involved in conflict are the most important. Try and give a proof of this.

Diagram of the traceback mechanism?

Quantifying the size of the branch pruned by an implied clause is important because not all pruning will significantly reduce the search space. Given the overhead of generating implied clauses, those clauses are unlikely to significantly improve runtime performance unless they prune large portions of the search

Table 1. Results from running Swan on the SAT-encoded CSP instance `direct0006`

Implied clauses	**Visible to heuris-tic**	**Time (s)**	**Branches**	**Unit prop-agations**	**Backtracks**
No	–	278.00	582234	21857718	291034
Yes	No	215.90	457883	14338664	228860
Yes	Yes	3.45	5460	220985	2623

space. Fig. 8 shows the start of a search space that might be explored by a DPLL solver, where denotes failure nodes in the search space, and nodes which only fail when implied clauses are added to the original formula. If an implied clause becomes false under the assignment at node a, then the implied clause is involved in the conflict but does not contribute to pruning the search space because node a was already a failure node. Even if an implied clause becomes false under the assignment at node b, the clause does not result in any significant pruning because the successors of b, nodes c and d, were already failure nodes. However, a significant portion of the search space would be pruned if node e became a failure node because of an implied clause.

Our implementations of both the conflict analysis and the branch pruning measurement take a significant amount time to compute, so we do not have results over a large set of benchmarks. The conflict analysis is fast enough that we could examine a number of examples

We will present two examples which appear to be typical of those instances which we could study in this way, and which... direct0006 and a qg instance.

4 Related work

Our current preprocessor implementation differs from 2CLS+EQ in one important respect: it does not perform the EqReduce operation described in [1]. While we could have modified 2CLS+EQ to operate as a preprocessor, including the EqReduce operation, doing so is not trivial and would limit the possibility of experiments examining the effect of adding EqReduce to HypBinRes. However, Bacchus has since written a preprocessor called HYPRE that combines HypBinRes and EqReduce. HYPRE can take a long time to run, but substantially improves solver performance on many problems. It can also solve some hard problems without search.

5 Future work

There is little existing work which analyses the reasons for the performance impact of a preprocessing algorithm, so we plan to investigate the interactions between inference and branching heuristics for other inference rules such as length bounded resolution and neighbour resolution. Other popular heuristics may exhibit different behaviour in the presence of additional clauses, so we will also study other heuristics.

We know that there is a relationship between the branching order of DPLL and the ordered resolutions that will prune the search space explored by DPLL [9]. By automating the comparison of the search spaces explored by Swan and combining it with the conflict analysis, we would be able to study the size of the branches pruned by additional clauses for a large set of benchmarks. This would give two pieces of information: whether large subtrees in the search space have been pruned, and if so, by which of the implied clauses. It may be possible to show a correlation between the branching order and the clauses which prune significant portions of the search space, and then to use that correlation to choose which implied clauses to generate and add to the formula.

Based on the density plots (Figures 3 and 4) it may be possible to automatically determine how many implied clauses to add based on the relative alterations to the literal counts in the formula. If this technique is effective, it may be useful as an automatic measure for the likely effect on relative solver performance resulting from other preprocessing techniques that alter the formula.

Reasoning about the search space explored by a solver with conflict clause recording is difficult, but it would be possible to add clause recording to Swan, and then show which implied clauses contribute to the conflict clause generation. This would make it possible to study the interaction between the clauses added during preprocessing by HypBinRes and those added during search by conflict recording.

6 Conclusion

The effect of adding implied clauses on SAT solver performance can strongly interact with the branching heuristic used. In particular, using the HypBinRes rule to add clauses can result in Chaff searching a larger search space, with a corresponding increase in the solution time.

By isolating the effect of the additional clauses on the heuristic, we have shown that a DPLL solver without conflict recording such as Swan performs best when the implied clauses are visible to the heuristic. While the variable performance exhibited by Chaff on HypBinRes preprocessed instances could be partially due to interaction with conflict clauses, we have also shown that much of the poor performance on these instances can be explained by the Chaff heuristic ignoring the clauses added by HypBinRes in favour of those generated by conflict clause recording.

7 Acknowledgements

We are grateful to Ian Gent for making available the CSP instances used in his paper, to Fahiem Bacchus and the authors of Chaff for making their SAT solvers publically available, to Dan Sheridan for his collaboration in developing the hints format, to Oliver Kullman for his suggestions, and to Toby Walsh and other members of the APES research group for their advice and encouragement. The first author is supported by EPSRC Grant GR/N16129 (for more information see http://www.cs.york.ac.uk/aig/projects/implied).

References

[1] Fahiem Bacchus. Extending Davis Putnam with extended binary clause reasoning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2002)*, 2002 (to appear).

[2] 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.

[3] S.A. Cook. The complexity of theorem-proving procedures. In *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] Johan de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of the 11th Joint International Conference on Artificial Intelligence (IJCAI-89)*, pages 290–296, 1989.

[7] DIMACS. Suggested satisfiability format. Available from ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.tex, May 1993.

[8] Lyndon Drake, Alan Frisch, Ian Gent, and Toby Walsh. Automatically reformulating SAT-encoded CSPs. In *Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems (at CP-2002)*, Cornell, 2002.

[9] Alan M. Frisch. Solving constraint satisfaction problems with NB-resolution. *Electronic Transactions on Artificial Intelligence*, 3:105–120, 1999.

[10] Ian P. Gent. Arc consistency in SAT. In *ECAI 2002: the 15th European Conference on Artificial Intelligence*, 2002.

[11] Pascal Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, 1992.

[12] Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the Year 2000*, volume 63 of *Frontiers in Artificial Intelligence and Applications*, pages 283–292. IOS Press, 2000.

[13] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 1997.

[14] João P. Marques Silva and Karem A. Sakallah. Conflict analysis in search algorithms for satisfiability. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, Nov 1996.

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, Las Vegas, June 2001.

[16] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[17] Allen van Gelder. Combining preorder and postorder resolution in a satisfiability solver. In Henry Kautz and Bart Selman, editors, *Electronic Notes in Discrete Mathematics*, volume 9. Elsevier Science Publishers, 2001.

[18] Toby Walsh. SAT v CSP. In *Proceedings of CP-2000*, LNCS-1894, pages 441–456. Springer-Verlag, 2000.

[19] Hantao Zhang and Jieh Hsiang. Solving open quasigroup problems by propositional reasoning. In *Proceedings of International Computer Symposium*, 1994.