Automatically Reformulating SAT-Encoded CSPs

Lyndon Drake[1], Alan M. Frisch[1], Ian Gent[2], and Toby Walsh[3]

[1] 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}

[2] School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SS, United Kingdom
ipg@dcs.st-and.ac.uk
http://www.dcs.st-and.ac.uk/~ipg

[3] Cork Constraint Computation Centre, University College Cork, Cork, Ireland
tw@4c.ucc.ie
http://www.4c.ucc.ie/~tw

Abstract. We examine two encodings of binary constraint satisfaction problems (CSPs) into propositional satisfiability (SAT). We show that hyper-resolution rules can be used to infer, from the direct encoding, many of the clauses in the better-performing support encoding. Our experimental results confirm that applying hyper-resolution in this way reduces both the runtime and the number of search nodes used by a SAT solver on the encoded CSPs.

1 Introduction

Propositional satisfiability (SAT) is the archetypal NP-complete problem [4]. It is possible to efficiently solve a wide range of problems, such as planning [10], quasigroup completion [15], and model checking [3], by mapping them into SAT and solving the SAT representation of the problem. Such SAT instances are called *structured* instances, and a great deal of recent SAT research, particularly on the performance of SAT solvers, has been driven by the the interest in solving such instances.

Constraint satisfaction problems (CSPs) can also be reformulated as SAT instances. This paper considers two encodings of binary CSPs into SAT—the *direct* encoding [14], and the *support* encoding [6]. We show that it is possible to infer clauses from the support encoding by applying hyper-resolution to the direct encoding. We also experimentally compare the performance of a SAT solver when support clauses are inferred before search or during search.

For convenience, in the examples below all n variables in the CSP have the same domain size d, and the SAT variable $x_{i,v}$ is true if the CSP variable i takes value v. Following [6] we only examine binary constraints.

$x_{a,1} \vee x_{a,2} \vee x_{a,3}$	$x_{b,1} \vee x_{b,2} \vee x_{b,3}$	$x_{c,1} \vee x_{c,2} \vee x_{c,3}$
$\neg x_{a,1} \vee \neg x_{b,1}$	$\neg x_{b,1} \vee \neg x_{c,1}$	$\neg x_{c,1} \vee \neg x_{a,1}$
$\neg x_{a,2} \vee \neg x_{b,2}$	$\neg x_{b,2} \vee \neg x_{c,2}$	$\neg x_{c,2} \vee \neg x_{a,2}$
$\neg x_{a,2} \vee \neg x_{b,1}$	$\neg x_{b,2} \vee \neg x_{c,1}$	$\neg x_{c,2} \vee \neg x_{a,1}$
$\neg x_{a,3} \vee \neg x_{b,3}$	$\neg x_{b,3} \vee \neg x_{c,3}$	$\neg x_{c,3} \vee \neg x_{a,3}$
$\neg x_{a,3} \vee \neg x_{b,2}$	$\neg x_{b,3} \vee \neg x_{c,2}$	$\neg x_{c,3} \vee \neg x_{a,2}$
$\neg x_{a,3} \vee \neg x_{b,1}$	$\neg x_{b,3} \vee \neg x_{c,1}$	$\neg x_{c,3} \vee \neg x_{a,1}$

Fig. 1. Direct encoding example

1.1 The Direct Encoding

The direct encoding involves two kinds of clauses: *at-least-one* clauses to ensure that each variable is assigned at least one value from its domain; and *conflict* clauses to prevent constraint violations. These clauses take the following forms:

– **At-least-one**

$$x_{i,1} \vee x_{i,2} \cdots \vee x_{i,d}$$

one clause of this form for each variable i, where $\{1, 2, \ldots, d\}$ is the domain of i.

– **Conflict**

$$\neg x_{i,v} \vee \neg x_{j,w}$$

one clause of this form for each pair of assignments, $i = v$ and $j = w$, that violate a constraint between i and j.

For example, say we have a CSP with three variables, a, b, and c, with the constraints $a < b$, $b < c$, and $c < a$, where each variable has the domain $\{1, 2, 3\}$. If we encode this CSP as a SAT instance using the direct encoding, we generate the clauses shown in Figure 1. It should be noted that at-most-one clauses, as described below, are not required in order to test satisfiability with the direct encoding, unless a one-to-one correspondence is required between solutions in the SAT instance and the original CSP instance.

1.2 The Support Encoding

Instead of encoding constraint conflicts, it is possible to encode the support for a particular value. This idea was introduced by Kasif [9], and expanded on by Gent [6]. The basis of the idea is that for the constraint $a < b$ the assignment of value 1 to a implies that b must be either 2 or 3. We say that for this constraint the *support* for $a = 1$ is the set $\{2, 3\}$ and we would capture this fact with the *support clause* $\neg x_{a,1} \vee x_{b,2} \vee x_{b,3}$.

In general, if C is a binary constraint between variables i and j then the support for $i = v$ is the set of values that can be assigned to j without violating C. Notice that this set could be empty. If $\{s_1, \ldots, s_n\}$ is the support for $i = v$, then the support clause for C with $i = v$ is $\neg x_{i,v} \vee x_{j,s_1} \vee \cdots \vee x_{j,s_n}$.

$\neg x_{a,3}$	$\neg x_{b,3}$	$\neg x_{c,3}$
$\neg x_{a,2} \vee x_{b,3}$	$\neg x_{b,2} \vee x_{c,3}$	$\neg x_{c,2} \vee x_{a,3}$
$\neg x_{a,1} \vee x_{b,2} \vee x_{b,3}$	$\neg x_{b,1} \vee x_{c,2} \vee x_{c,3}$	$\neg x_{c,1} \vee x_{a,2} \vee x_{a,3}$
$\neg x_{b,1}$	$\neg x_{c,1}$	$\neg x_{a,1}$
$\neg x_{b,2} \vee x_{a,1}$	$\neg x_{c,2} \vee x_{b,1}$	$\neg x_{a,2} \vee x_{c,1}$
$\neg x_{b,3} \vee x_{a,1} \vee x_{a,2}$	$\neg x_{c,3} \vee x_{b,1} \vee x_{b,2}$	$\neg x_{a,3} \vee x_{c,1} \vee x_{c,2}$

Fig. 2. Example of support clauses

The support encoding of a CSP requires the at-least-one clauses, support clauses, and at-most-one clauses (to ensure that no CSP variable is assigned more than one value at once). The at-least-one clauses are identical to those in the direct encoding, while the support and at-most-one clauses take the following forms:

- **Support**

  $$x_{i,v_1} \vee x_{i,v_2} \cdots \vee x_{i,v_k} \vee \neg x_{j,w}$$

  one clause of this form for every constraint between variables i and j, for every value w in the domain of j, where $v_1, v_2, \ldots, v_k$ are the supporting values in variable i for $j = w$.
- **At-most-one**

  $$\neg x_{i,v} \vee \neg x_{i,w}$$

  one clause of this form for every variable i and pair (v, w), where $1 \leq v < w \leq d$.

Figure 2 show the clauses that encode the support for the example CSP described in the previous section.

Gent [7] points out that it is possible to use a series of binary resolution steps to infer the support clauses from the direct encoding. Below, we show that hyper-resolution can be used to perform the conversion, taking only a single reasoning step to infer each support clause.

1.3 Hyper-resolution

The most well-known form of resolution, binary resolution [12], infers a new clause from two parent clauses. Hyper-resolution can be treated as a single inference step that makes the same inference as several binary resolution steps.[4] One advantage of a well-chosen hyper-resolution rule over repeated application

[4] In fact, the definition of hyper-resolution given in [13] is much stricter than this, and in particular requires an ordering over the clauses and limits the signs of literals involved in the hyper-resolution. For the sake of convenience we follow Bacchus' terminology.

of binary resolution is that fewer extraneous clauses are generated, as the intermediate clauses are ignored [13, p. 218].

Bacchuss [1] has shown that one such rule, which he calls HypBinRes, can improve the performance of a DPLL [5] search implementation. The SAT solver 2CLS+EQ [2] is an implementation that combines HypBinRes and DPLL search, and appears to be the only publicly available solver that applies complex reasoning during search while remaining competitive with the fastest available solvers. The HypBinRes hyper-resolution rule is defined as follows:

Definition 1 (HypBinRes). *Assuming that literals in clauses can be reordered, given a non-unit clause of the form* $(x_1 \vee x_2 \vee \cdots \vee x_n)$*, and* $n-1$ *binary clauses of the form* $(\neg x_i \vee h)$ *where* $1 \leq i < n$*, infer the clause* $(x_n \vee h)$*.*

For example, if we apply HypBinRes to the clauses $(x_1 \vee x_2 \vee x_3 \vee x_4)$, $(\neg x_1 \vee h)$, $(\neg x_2 \vee h)$, and $(\neg x_4 \vee h)$, the clause $(x_3 \vee h)$ is inferred.

2 Inferring Support Clauses from the Direct Encoding

Below we consider two methods for automatically inferring support clauses from the direct encoding: the first infers some of the support clauses, while the second infers all support clauses.

2.1 Inferring Some Support Clauses

Our first observation is that applying the HypBinRes rule given in Definition 1 to the direct encoding of a CSP infers a subset of the support clauses. For example, on the constraint $a < b$ with domain $\{1, 2, 3\}$ the at-least-one clause for a is $(x_{a,1} \vee x_{a,2} \vee x_{a,3})$, and the conflict clauses for $b = 2$ are $(\neg x_{a,2} \vee \neg x_{b,2})$ and $(\neg x_{a,3} \vee \neg x_{b,2})$. If we apply HypBinRes to these three clauses, we derive $(x_{a,1} \vee \neg x_{b,2})$, which is the corresponding support clause. However, only some of the support clauses can be inferred using HypBinRes.

Theorem 1. *From the at-least-one clause for* i*, and the conflict clauses for* i *where* $j = w$*, the HypBinRes rule infers the corresponding support clause for* $j = w$ *if and only if that support clause is binary.*

Proof. ($\Longleftarrow$) From the description of the direct encoding, we know that the at-least-one clause for i is of the form $(x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,d})$, where d is the domain size. If the support clause is binary, then only one value v_a in the domain of i is allowed. So there are $d-1$ conflict clauses of the form $(\neg x_{i,v_k} \vee \neg x_{j,w})$ where $1 \leq k \leq d$, $k \neq a$ and i, j, w are constant. By the HypBinRes rule, we can infer $(x_{i,v_a} \vee \neg x_{j,w})$ from the conflict clauses and the at-least-one clause, which is the appropriate support clause for $j = w$.

($\Longrightarrow$) It is obvious that if a support clause is non-binary, then the conditions for applying HypBinRes are not met, and so only binary support clauses can be inferred by HypBinRes. $\square$

Interestingly, if all the support clauses for a given constraint are binary, then the constraint is functional [8]. In other words, HypBinRes can infer all the support clauses for a functional constraint.

2.2 Inferring All Support Clauses

HypBinRes requires $n-1$ binary clauses before it can infer a support clause. If we weaken the precondition of HypBinRes such that only m binary clauses, $m \leq n$, are required, then all the support clauses can be inferred from the direct encoding by this new hyper-resolution rule.

For example, on the constraint $a < b$, where both variables have the domain $\{1, 2, 3\}$ the at-least-one clause for a is $(x_{a,1} \vee x_{a,2} \vee x_{a,3})$ and the only conflict clause for $b = 1$ is $(\neg x_{a,1} \vee \neg x_{b,1})$. While HypBinRes cannot be applied to these clauses, we can apply Generalised HypBinRes to infer $(x_{a,2} \vee x_{a,3} \vee \neg x_{b,1})$, which is the corresponding support clause for $b = 1$. (This hyper-resolution is also a binary resolution, but the point is that the Generalised HypBinRes rule can make this inference while the original HypBinRes rule cannot.)

Definition 2 (Generalised HypBinRes). *Assuming that literals in clauses can be reordered, from a non-unit clause of the form $(x_1 \vee x_2 \vee \cdots \vee x_n)$, and m binary clauses of the form $(\neg x_i \vee h)$ where $1 \leq i \leq m < n$ infer the clause $(x_{m+1} \vee \cdots \vee x_n \vee h)$.*

Theorem 2. *Given the at-least-one clause for i, and the set of conflict clauses for $j = w$, applying the hyper-resolution rule given in Definition 2 infers the corresponding support clause for $j = w$.*

Proof. Let V be the set containing precisely the values of i that conflict with $j = w$ and let m be the cardinality of V. Then the conflict clauses for the constraint include $\{\neg x_{i,v} \vee \neg x_{j,w} \mid v \in V\}$. The at-least-one clause for i is $(\alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_d)$, where $(\alpha_1, \alpha_2, \ldots, \alpha_d)$ is an ordering over the literals $(x_{i,1}, x_{i,2}, \ldots, x_{i,n})$ such that $V = \{\alpha_1, \ldots, \alpha_m\}$. Applying the hyper-resolution rule given in Definition 2 (substituting $\neg x_{j,w}$ for h) infers the clause $(\alpha_{m+1} \vee \cdots \vee \alpha_d \vee \neg x_{j,w})$, which is the support clause for $j = w$. □

It is worth noting that even this more general rule cannot infer non-support clauses, and in particular, it cannot infer the at-most-one clauses (as the at-most-one clauses are not implied by the other clauses in the formula). As the parent clauses in these hyper-resolutions are not deleted from the formula, the at-most-one clauses are not required for the encoding to be correct.

It is not clear that the extra support clauses inferred by the Generalised HypBinRes rule would actually improve the performance of a solver on an instance. One of the motivations for inferring binary clauses is that they are particularly beneficial to a DPLL search based solver, so inferring non-binary clauses may not be the most beneficial choice.

2.3 Inferring Support Clauses During Search

The 2CLS+EQ solver not only applies HypBinRes before search, but also whenever possible during search. Applying HypBinRes during search has the effect of inferring a support clause whenever a particular value in the domain of i is supported by only one value in the domain of j. In other words, 2CLS+EQ only infers a support clause at the point where the support clause is binary, but is likely to infer many more support clauses than simply applying HypBinRes to the direct encoding before search.

3 Experimental Results

We compared the performance of 2CLS+EQ on the direct encoding when HypBinRes was applied only during preprocessing to its performance when HypBinRes was applied during search. The test instances used were the same hard CSP instances used by Gent [6]. Applying HypBinRes during search massively reduced both runtimes (see Figure 3) and the number of search nodes explored. Runtimes were reduced by a median of 100 seconds (about 90% of the median preprocessing runtime), while the number of search nodes was typically reduced from tens of thousands to less than a hundred. Very few hyper-resolutions were found during preprocessing, indicating that the test instances contained very few functional constraints, but many hyper-resolutions were identified during search. What is more, the additional hyper-resolutions discovered during search resulted in substantially improved performance.

We also looked at the performance of 2CLS+EQ on the support encoding, where enabling HypBinRes during search provided only a small improvement over restricting HypBinRes to preprocessing. Surprisingly, several thousand hyper-resolvents were identified during preprocessing on most of the support encoding instances; presumably these inferences involve the at-most-one clauses. Even more interesting is that fact that 2CLS+EQ (with HypBinRes during search) on the direct encoding outperforms 2CLS+EQ on the support encoding by a factor of two.

It is worth noting that Chaff [11] is still considerably faster than 2CLS+EQ on both encodings, and that Chaff performs much better on the support encoding than on the direct encoding.

All the experiments were carried out on dual-processor Pentium III 750 PCs with at least 512MB of RAM (none of the solvers exceeded the available memory during search), running Linux 2.4.16.

4 Future Work

The Generalised HypBinRes rule we gave in Definition 2 to infer all the support clauses may be useful on encodings of CSPs and other problems. At the very least, it would be worthwhile to experimentally test the effect of applying this

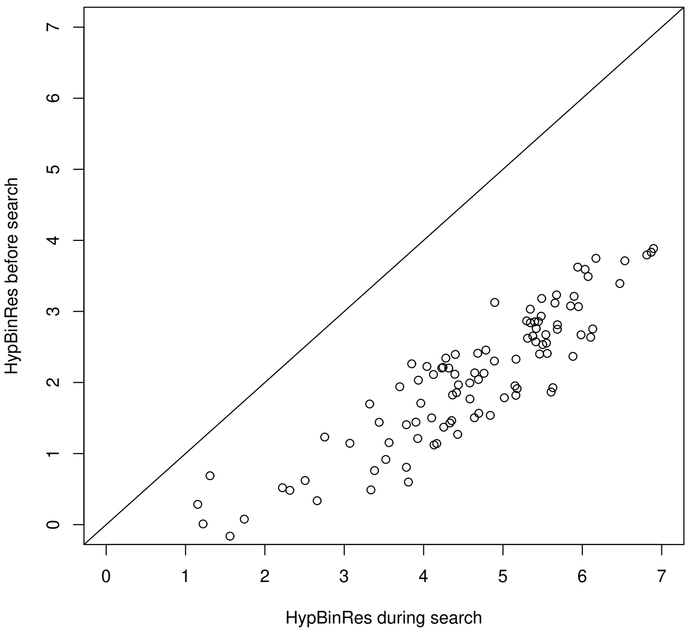

Fig. 3. Comparison of 2CLS+EQ runtimes (in seconds) with HypBinRes enabled during preprocessing and during search.

rule to the direct encoding to determine whether inferring non-binary support clauses improves the performance of DPLL search on those instances.

One important difference between the clauses in the support encoding and those generated by 2CLS+EQ is that 2CLS+EQ adds support clauses without removing any of the original clauses, while the support encoding omits the conflict clauses and adds at-most-one clauses. It would be interesting to determine whether or not the conflict clauses and the at-most-one clauses improve search performance – in other words, would a combination of clauses from both encodings be better?

A third area of interest is using HypBinRes as a preprocessing step. Chaff is much faster than 2CLS+EQ on these instances, despite applying less inference and as a result having to explore many more search nodes. An efficient implementation of conflict learning in a SAT solver requires the use of lazy data structures [11,16]. The challenge is to create a solver that combines complex reasoning during search with efficient conflict learning in order to outperform an efficient existing implementations of DPLL with conflict learning. One way around this problem is to perform inference as a preprocessing step, so we intend to study the use of HypBinRes as a preprocessor for conventional SAT solvers such as Chaff.

Finally, this paper explains some of the effects of applying HypBinRes to the direct encoding of CSPs into SAT. Published results show that HypBinRes appears to be beneficial to DPLL search on mappings of other problems into SAT, so we hope to identify similar explanations for other such problem encodings.

5 Conclusions

DPLL-style SAT solvers perform better on the support encoding of CSPs into SAT than on the direct encoding. The HypBinRes rule can be used to automatically infer some of the clauses provided by the support encoding, and experimental results show that these inferred support clauses improve SAT solver performance.

The most interesting point to remember is that HypBinRes was devised in order to improve the performance of a DPLL-based SAT solver, not to convert between competing CSP encodings. The fact that HypBinRes carries out part of the conversion from one CSP encoding to a newer, superior encoding is a validation of its usefulness, and an encouragement for further research in the area.

6 Acknowledgements

We are grateful to Fahiem Bacchus for making available both a preprint of his AAAI paper and the 2CLS+EQ solver. The first author is supported by EPSRC Grant GR/N16129 (see http://www.cs.york.ac.uk/aig/projects/implied). The third author is supported by EPSRC Grants GR/R29666, GR/R55382, and GR/M90641.

References

1. Fahiem Bacchus. Exploring the computational tradeoff of more reasoning and less searching. In *Fifth International Symposium on Theory and Applications of Satisfiability Testing*, pages 7–16, 2002.
2. Fahiem Bacchus. Extending Davis Putnam with extended binary clause reasoning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2002)*, pages 613–619, 2002.
3. 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.
4. S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*, pages 151–158, 1971.
5. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
6. Ian P. Gent. Arc consistency in SAT. In *ECAI 2002: The 15th European Conference on Artificial Intelligence*, 2002.
7. Ian P. Gent. Arc consistency in SAT. Technical Report APES-39A-2002, APES Research Group, January 2002.
8. 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.
9. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45:275–286, 1990.
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. 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.
12. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.
13. J. A. Robinson. *Logic: Form and Function.* Artificial Intelligence Series. North-Holland, 1979.
14. Toby Walsh. SAT v CSP. In *Proceedings of CP-2000*, LNCS-1894, pages 441–456. Springer-Verlag, 2000.
15. Hantao Zhang and Jieh Hsiang. Solving open quasigroup problems by propositional reasoning. In *Proceedings of International Computer Symposium*, 1994.
16. Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putman method. 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 309–326. IOS Press, 2000.