

A Simplifier for Propositional Formulas with Many Binary Clauses

Ronen I. Brafman

Department of Computer Science

Ben-Gurion University

Beer-Sheva, Israel

brafman@cs.bgu.ac.il

Abstract

Deciding whether a propositional formula in conjunctive normal form is satisfiable (SAT) is an NP-complete problem. The problem becomes linear when the formula contains binary clauses only. Interestingly, the reduction to SAT of a number of well-known and important problems – such as classical AI planning and automatic test pattern generation for circuits – yields formulas containing many binary clauses. In this paper we introduce and experiment with 2-SIMPLIFY, a formula simplifier targeted at such problems. 2-SIMPLIFY constructs the implication graph corresponding to the binary clauses in the formula and uses this graph to deduce new unit literals. The deduced literals are used to simplify the formula and update the graph, and so on, until stabilization. Finally, we use the graph to construct an equivalent, simpler set of binary clauses. Experimental evaluation of this simplifier on a number of bench-mark formulas produced by encoding AI planning problems prove 2-SIMPLIFY to be fast and effective.

1 Introduction

Propositional satisfiability (SAT) is the problem of deciding whether a propositional formula in conjunctive normal form (CNF) is satisfiable. SAT was the first problem shown to be NP-complete [Cook, 1971] and has important practical applications. In the last decade we have witnessed great progress in SAT solution methods, first with the introduction of efficient stochastic local search algorithms [Selman *et al.*, 1992; 1994], and more recently with a number of efficient systematic solvers, such as REL-SAT [Bayardo and Schrag, 1997] and SATZ [Li and Anbulagan, 1997] and their randomized versions [Gomes *et al.*, 1998].

Among AI researchers, interest in SAT solution algorithms has increased farther since Kautz and Selman showed that some classical planning problems can be solved more quickly when they are reduced to SAT problems [Kautz and Selman, 1996]. Kautz and Selman's *planning as satisfiability* approach is based on generic SAT technology, and, aside from the translation process itself, makes no use of properties specific to planning problems. However, SAT-encoded planning

problems have an important syntactic property: they contain a large fraction of binary clauses. Interestingly, this property is found in another important domain – automatic test-pattern generation for circuits [Larrabee, 1992].

Unlike the general SAT problem, 2-SAT, the problem of deciding whether a propositional formula containing binary clauses *only* has a satisfying assignment is (constructively) solvable in linear time. One would hope that this property would make it easier to solve SAT instances containing a large fraction of binary clauses. In this article we describe the 2-SIMPLIFY preprocessor, a simplifier that is geared to such formulas. Like other simplifiers (e.g., Crawford's COMPACT), this algorithm takes a propositional formula in CNF as input and outputs a new propositional formula. Naturally, for this process to be worthwhile, the new formula should be easier to solve, and the overall time required for simplification and solution of the simplified formula should be less than the time required to solve the original SAT instance. Experiments on a number of bench-mark formulas derived from planning problems show that 2-SIMPLIFY meets this criteria.

2-SIMPLIFY efficiently implements and combines well-known 2-SAT techniques, a limited form of hyper-resolution, and novel use of transitive reduction to reduce formula size. The basic idea is as follows: each clause of the form $p \vee q$ is equivalent to two implications: $\neg p \rightarrow q$ and $\neg q \rightarrow p$. We use this property to construct a graph (known as the *implication graph* [Aspvall *et al.*, 1979]) from the set of binary clauses. This graph contains a node for each literal in the language and a directed edge from a literal l to another literal l' if the (disjunction equivalent to the) implication $l \rightarrow l'$ appears among the set of binary clauses. After constructing this graph, we compute its transitive closure and check each literal to see whether its negation appears among its descendants. If $\neg l$ is a descendant of l , we can immediately conclude that $\neg l$ is a consequence of the original formula. Once we know that l holds, we can simplify the original formula. The simplified formula may contain new binary clauses, which are immediately added to the graph. 2-SIMPLIFY utilizes these and other ideas to quickly derive unit literals from the original formula and produce the simplest equivalent formula as its output.

In the next section we discuss the background of this work in more detail. In Section 3 we present the simplification algorithm used by 2-SIMPLIFY. In Section 4 we present some experimental results, and we conclude in Section 5.

2 Background

We start with some SAT background and then we briefly explain the *planning as satisfiability* approach, which motivated this work.

2.1 The SAT Problem

The SAT problem is defined as follows: given a propositional formula in conjunctive normal form (CNF) output YES if the formula is satisfiable and NO otherwise. In practice, a positive answer is accompanied by some satisfying assignment.

There are two classes of SAT algorithms: stochastic and systematic. Stochastic methods, such as G-SAT[Selman *et al.*, 1992] and WALKSAT[Selman *et al.*, 1994], perform stochastic local search in the space of truth assignment. Often, they can find solutions quickly, but they cannot identify an unsatisfiable instance. Their performance is extremely sensitive to the choice of heuristic and various other parameters. Systematic methods systematically search the space of truth assignments. Thus, they can identify unsatisfiable instances. Modern systematic algorithms are quite fast and stable thanks to improved branch choice heuristics and backtracking techniques. In addition, systematic solvers can be improved by introducing some randomization into their search procedure, e.g., their choice of branch variable. See [Gomes *et al.*, 1998] for more information on this topic.

Often, a formula simplifier is applied before the SAT solver. Simplifiers use specialized, efficient deductive methods to reduce the original formula into a simpler formula which is typically easier to solve. The best known simplification method is unit propagation. When one of the clauses in the formula contains a single literal, it must be assigned the value *true* in any satisfying assignment. For example, if a formula contains the clause $\{\neg p\}$ then $\neg p$ must be *true* in any satisfying assignment, i.e., p must be *false*. Once we deduce this fact, we can use it to simplify other clauses: clauses that contain $\neg p$ can be removed since their satisfaction is guaranteed when p is *false*, and the literal p can be removed from any clause containing it (e.g., $\{p, \neg s\}$ will be transformed into $\{\neg s\}$) because it is equivalent to *false*. As we just saw, the simplification process can yield additional unit clauses, which are used to produce additional simplifications. If during the simplification process an empty clause is discovered (e.g., if we assigned s the value *true* and there is a unit clause $\{\neg s\}$) we can conclude that the formula is unsatisfiable.

There are a number of additional simplification methods, such as failed unit clause and failed binary clause, where a unit or binary clause are added to the current formula and we attempt to show (e.g., using unit propagation) that the resulting formula is inconsistent. In that case, the negation of the added clause is implied by the original formula, and we update the truth assignment accordingly. For example, if our original formula becomes inconsistent once we add the clause $\{p, q\}$, we know that both p and q must be assigned *false*.

2.2 2-SAT

2-SAT is a subclass of SAT in which clauses contain no more than two literals. While SAT is NP-complete [Cook, 1971], 2-SAT can be solved in linear time. The key step in solving 2-SAT problems is the construction of the *implication*

Instance	% Binary Clauses
log-dir.a	49%
log-dir.b	55%
log-dir.c	55%
log.d	80%
log-gp.a	98%
log-gp.b	98%
log-gp.c	98%
log-un.a	98%
log-un.b	98%
log-un.c	99%
bw-dir.a	70%
bw-dir.b	71%
bw-dir.c	74%
bw-dir.d	78%

Figure 1: Percentage of Binary Clauses in SAT-Encoded Planning Problems

graph [Aspvall *et al.*, 1979]). The nodes of the implication graph correspond to the literals in the formula. The graph contains an edge between the literal l and the literal l' if the clause $\{\neg l, l'\}$ appears in the formula. That is, edges in the graph correspond to implications (since $\{\neg l, l'\}$ is equivalent to $l \rightarrow l'$). Since implication is transitive, we have that $l_1 \rightarrow l_2$ is implied by the formula whenever there is a path in the graph between node l_1 and node l_2 . In particular, if we have a path between l_1 and $\neg l_1$, we know that l_1 cannot hold. Therefore, $\neg l_1$ is implied by the formula. If, in addition, we have a path from $\neg l_1$ to l_1 , then neither l_1 nor $\neg l_1$ can hold, and so the formula is unsatisfiable. Finally, we know that in every satisfying truth assignment, if l is assigned *true* then any literal implied by l , i.e., any descendant of l in the graph, must be assigned *true* as well.

2.3 Planning As Satisfiability

The planning problem is defined as follows: given a description of an initial state, a goal state, and a set of operators (=Actions) for changing the state of the world, find a sequence of operators that, when applied in the initial state, yield the goal state. An important development in planning algorithms was Kautz and Selman's *planning as satisfiability* approach [Kautz and Selman, 1996]. Kautz and Selman showed that by reducing planning problems to satisfiability problems, we can often solve them more quickly than by using standard planning algorithms. Planning problems can be encoded as satisfiability problems in a number of ways (e.g., see [Ernst *et al.*, 1997] for a description and analysis of some of these methods).

As [Brafman, 1999] points out, encoded planning problems contain a large number of binary clauses. In Figure 1 we show this for a number of instances of SAT-encoded planning problems.¹ This is no accidental phenomenon. Close inspection of the types of constraints expressed within encoded planning problems makes it apparent that many classes of these constraints generate binary formulas. For example,

¹ Available at <ftp://ftp.research.att.com/dist/ai/logistics/tar.Z> and <satplan.data.tar.Z>

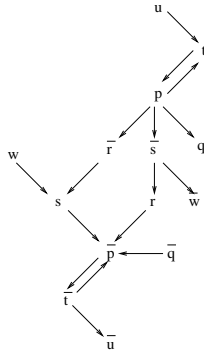


Figure 2: The Implication Graph

the constraint that if an action is executed at some time point then all its preconditions must hold prior, produces binary clauses. Similarly, the constraint asserting that if an action is executed at some point then all its effects must hold after the execution yields binary clauses as well. In the encoding used by the BLACKBOX planner [Kautz and Selman, 1999] mutual exclusion constraints (on actions and on state variables) play a prominent role. These constraints are expressed using binary clauses as well.

Interestingly, it turns out that the SAT encodings of other important problems exhibit the same large percentage of binary clauses. These include test-pattern generation for circuits [Larrabee, 1992] and bounded model checking [Shtrichman, 2000].

3 The 2-SIMPLIFY PreProcessor

We now explain the algorithm implemented by the 2-SIMPLIFY preprocessor using the following formula:

$$\begin{aligned} &\{\neg p, q\}, \{\neg p, \neg r\}, \{r, s\}, \{\neg w, s\}, \{\neg p, t\}, \{\neg t, p\}, \{\neg u, t\} \\ &\{\neg p, \neg s\}, \{\neg p, s, q\}, \{\neg q, \neg s, p\}, \{u, \neg w, q\}, \{\neg q, \neg r, s\} \\ &\{s, v, \neg m\} \end{aligned}$$

(1) *Construct Implication Graph.* A graph containing all literals in the language is constructed with directed edges from l to l' if $\{\neg l, l'\}$ is a binary clause. Figure 2 shows the implication graph for the formula above.

(2) *Collapse Strongly Connected Components.* A subgraph in which there is a path between every pair of nodes is called a strongly connected component (SCC). When a path from node l to node l' exists, we know that $l \rightarrow l'$ is a consequence of our formula. Therefore, all nodes within an SCC imply each other, and they must all be assigned the same value.

Once we discover an SCC we replace it by a single node. The children of this node are the children of the nodes in the SCC, and the parents of this node are the parents of the nodes in this SCC. In addition, all literals in the SCC must be replaced by this new node within all non-binary clauses.

In our example, the nodes t and p , and the nodes $\neg t$ and $\neg p$, form strongly connected components. We choose p to represent the first SCC and we choose $\neg p$ to represent the second SCC. The reduced graph is shown in Figure 3.

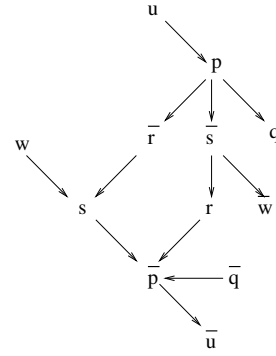


Figure 3: Removing Strongly Connected Components

(3) *Generate Transitive Closure.* We generate the transitive closure. Now, we know that if l' is a child of l then $l \rightarrow l'$ is implied by the original formula. We can deduce l if either:

1. for some proposition p , both p and $\neg p$ are children of $\neg l$.
2. l is a child of $\neg l$.

Once we deduce l , we can perform unit propagation: We know that all children of l are true, and we can reduce the current formula by applying unit propagation. If the reduced formula contains new binary clauses, we add the appropriate edges to the graph and update the transitive closure.

In Figure 4 we can see the effect of this step. First, we compute the transitive closure of the current graph, shown in Figure 4A. In this graph, we see that u has $\neg u$ as a descendant and that p has $\neg p$ as a descendant. Therefore, we conclude that p and u must be assigned the value *false*. We can remove nodes that correspond to assigned propositions (i.e., $p, u, \neg p, \neg u$ in our case). The resulting graph is shown in Figure 4B. Next, we perform unit propagation, and our initial ternary clauses: $\{\neg p, s, q\}, \{\neg q, \neg s, p\}, \{u, \neg w, q\}, \{\neg q, \neg r, s\}$ are reduced to $\{\neg q, \neg s\}, \{\neg w, q\}, \{\neg q, \neg r, s\}$. The first clause was removed because it is satisfied, and a (false) literal was removed from the next two clauses. Since we have new binary clauses, we can update the graph, as shown in Figure 4C, making sure it is transitively closed. In the resulting graph, $\neg w$ is a child of w , and we can deduce that $w = \text{false}$. The reduced graph is shown in Figure 4D.

(4) *Derive Shared Implications.* Let $\{l_1, \dots, l_k\}$ be some non-binary clause in the formula. Let L_i be the set of literals implied by l_i for $i = 1, \dots, k$. Let $L = L_1 \cap \dots \cap L_k$. All literals in L are consequences of our formula, and we can use them to perform unit propagation.

Consider the clause $\{\neg q, \neg r, s\}$, the sets of literals implied by each of the literals in this clause are $(\neg q), (\neg r, s, \neg q), (s, \neg q)$, respectively. Their intersection contains $\neg q$. Hence, we can deduce that q is *false*.

(5) *Compute Transitive Reduction.* The transitive reduction of a graph G is a graph G' with the same nodes as G but with a minimal set of edges such that a path between l and l' exists in G iff a path between l and l' exists in G' .

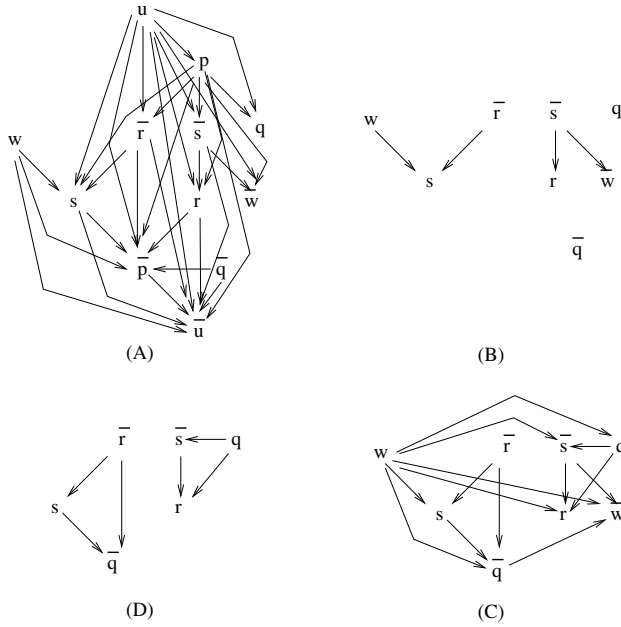


Figure 4: (A) Initial Transitive Closure. (B) Removal of Assigned Nodes. (C) Update with New Binary Clauses. (D) Removal of Assigned Nodes.

(6) *Output Simplified Formula.* We output a formula whose clauses consist of the non-binary clauses remaining after all simplifications were performed and all binary clauses corresponding to edges in the transitive reduction of the graph. Given the assignments deduced so far and the mappings between elements of strongly connected components, the simplified formula is equivalent to the original formula. For example, the output for our original formula will be: $\{r, s\}, \{s, v, \neg m\}$ together with the partial assignment $u = \text{false}, p = \text{false}, w = \text{false}, q = \text{false}$.

Step (4) is a novel implementation of an old technique (hyper-resolution [Robinson, 1965]) and step (5) is new. Both have important impact on 2-SIMPLIFY's performance. The *Derive Shared Literals* step enhances the ability of 2-SIMPLIFY to derive unit literals. In some cases, it can derive hundreds of new unit literals quickly. In fact, 2-SIMPLIFY uses a more sophisticated version of this procedure: if no shared unit literals exist, we attempt to derive new binary clauses by intersecting the implications of all literals but one. These binary clauses are then added to the implication graph. The *Compute Transitive Reduction* step leads to a minimal sufficient set of binary clauses, leading to smaller and simpler formulas. We have found this reduction to have an important positive influence on systematic solvers.

4 Experimental Evaluation

We evaluated the 2-SIMPLIFY preprocessor on a set of benchmark instances of encoded planning problems which were used to test the REL-SAT solver. 2-SIMPLIFY is written in C++ and all experiments described here were conducted on a DELL Latitude CPi notebook with a Pentium II-400 pro-

Instance	Simp. Time	Assigned/Total
log-dir.a	0.02	152/828
log-dir.b	0.02	152/843
log-dir.c	0.04	186/1141
log.d	2.4	753/4713
log-gp.a	0.19	148/1782
log-gp.b	0.31	169/2069
log-gp.c	0.52	191/2809
log-un.a	0.13	160/1415
log-un.b	0.21	161/1729
log-un.c	0.37	179/2353
bw-dir.a	0.08	173/459
bw-dir.b	0.2	351/1087
bw-dir.c	0.96	824/3016
bw-dir.d	3.86	1658/6325

Figure 5: Running time and deduction power of 2-SIMPLIFY

Instance	SATZ	2-Simplify	SATZ on 2s	Total
log-dir.a	108.03	0.02	0.51	0.53
log-dir.b	0.4	0.02	0.38	0.4
log-dir.c	3.23	0.04	1.11	1.15
log.d	1636.7	2.4	1.41	3.81
log-gp.a	0.95	0.19	0.17	0.36
log-gp.b	2.82	0.31	0.6	0.91
log-gp.c	4.08	0.52	0.8	1.32
log-un.a	0.5	0.13	0.09	0.22
log-un.b	UA	0.21	UA	UA
log-un.c	UA	0.37	UA	UA
bw-dir.a	0.2	0.08	0.08	0.16
bw-dir.b	0.7	0.2	0.32	0.52
bw-dir.c	3.27	0.96	1.79	2.75
bw-dir.d	1061.83	3.86	300.72	304.58

Figure 6: Solution times for SATZ and 2-SIMPLIFY+SATZ.

cessor with 64MB RAM running LINUX. All time measurements refer to CPU time.

First, we examined 2-SIMPLIFY's ability to deduce unit literals. In Figure 5 we show 2-SIMPLIFY's running time on each of the instances and the number of variables it was able to assign. Note that in addition to deducing unit literals, 2-SIMPLIFY also supplies additional important information in the form of equivalent literals.

To assess the utility of 2-SIMPLIFY we generated simplified formulas for each of the instances and compared the solution time of the original formulas with the combined simplification and solution times for the simplified formulas. We performed this comparison using two systematic solvers: SATZ and REL-SAT. The results for SATZ are shown in Figure 6, where we show the simplification time again, in order to give a sense of its magnitude in comparison to solution time. We see that in *all* instances 2-SIMPLIFY proves itself useful, reducing the overall solution time. 2-SIMPLIFY is particularly useful on the instances on which SATZ takes longest.

The results for REL-SAT are shown in Figure 7. We note that in many cases, REL-SAT's performance on the simplified formulas can be improved by disabling its own preprocessor.

Instance	REL-SAT	2-SIMPLIFY	2S+R
log-dir.a	0.46	0.02	0.41
log-dir.b	0.47	0.02	0.45
log-dir.c	1.04	0.04	0.8
log.d	9.64	2.4	9.47
log-gp.a	1.43	0.19	0.86
log-gp.b	2.22	0.31	1.56
log-gp.c	3.35	0.52	2.93
log-un.a	0.57	0.13	0.33
log-un.b	0.57	0.13	0.33
log-un.b	0.57	0.13	0.33
log-un.b	12.21	0.21	6.74
log-un.c	23.14	0.37	12.09
bw-dir.a	0.25	0.08	0.18
bw-dir.b	1.58	0.2	1.1
bw-dir.c	29.05	0.96	18.6
bw-dir.d	>1040	3.86	528.73

Figure 7: Solution times for REL-SAT, 2-SIMPLIFY, and 2-SIMPLIFY+REL-SAT.

(We refer the reader to the full paper where these experiments appear.) We see that 2-SIMPLIFY+REL-SAT is always faster than REL-SAT alone when we use REL-SAT’s preprocessor. The improvement is especially significant in the hardest instances. We note that the REL-SAT figures represent average running times (because REL-SAT has a stochastic element).²

We run another sequence of experiments to compare 2-SIMPLIFY with Crawford’s COMPACT simplifier. First, we examined the performance of various COMPACT options on the test problems and found that the best performance is obtained almost always using either no flags or using the *psl* flags.³ To see whether 2-SIMPLIFY improves on COMPACT’s performance we compare the sum of simplification time and solution time for combinations of COMPACT with and without 2-SIMPLIFY and with either SATZ or REL-SAT.

In Figure 8 we see the results for SATZ. The columns correspond to the combined running time of SATZ and the simplification algorithms on each of the instances. The first column is SATZ applied to COMPACT simplified formulas, while the second is SATZ applied to 2-SIMPLIFY+COMPACT. The third and forth columns are similar, except that we used the *psl* options in COMPACT. We see that 2-SIMPLIFY leads to reduced running times in all cases except two (log-dir.a and log.d on compact with the *psl* options).

In Figure 9 we show the corresponding results for REL-SAT (with its preprocessor). Here we see that 2-SIMPLIFY improves the overall performance on *all* problems, and in some cases significantly so.

Finally, we examined 2-SIMPLIFY’s influence on the per-

²In the case of bw-dir.d, REL-SAT timed out on the original problem in some of the iterations and we provided a lower bound on its average running time.

³With no flags, COMPACT does unit resolution, removes satisfied clauses, and renames variables to be contiguous. With *psl*, in addition to the above, COMPACT eliminates pure literals, resolves away variables with a single occurrence, and for each literal checks whether its addition leads to contradiction.

Instance	C+S	2S+C+S	psl + S	psl + 2S+ S
bw-dir.a	0.29	0.25	0.16	0.16
bw-dir.b	0.84	0.67	1.0	0.89
bw-dir.c	3.39	3.32	6.30	4.69
bw-dir.d	686.66	302.47	123.33	77.31
log-dir.a	1.17	0.77	2.29	13.35
log-dir.b	0.52	0.53	0.64	0.41
log-dir.c	1.32	3.94	398.33	3.23
log.d	454.76	4.20	4.06	5.39
log-gp.a	1.02	0.48	5.82	1.60
log-gp.b	1.59	0.76	10.24	2.25
log-gp.c	3.66	1.13	22.82	13.82
log-un.a	0.62	0.32	3.18	0.88
log-un.b	-	-	12.70	3.62
log-un.c	-	-	104.48	21.00

Figure 8: Simplification with COMPACT and 2-SIMPLIFY, Solution with SATZ.

Instance	C+R	2S+C+R	psl + R	2S+ psl + R
bw-dir.a	0.38	0.16	0.16	0.16
bw-dir.b	1.55	1.24	1.47	1.38
bw-dir.c	25.93	18.47	28.33	17.64
bw-dir.d	-	608.27	-	510.59
log-dir.a	0.45	0.52	0.56	0.38
log-dir.b	0.80	0.59	0.71	0.49
log-dir.c	1.43	1.13	1.21	0.79
log.d	10.96	9.33	9.32	8.63
log-gp.a	1.62	0.98	6.56	1.80
log-gp.b	2.40	1.54	12.16	3.01
log-gp.c	4.46	3.23	27.25	5.75
log-un.a	0.82	0.44	3.35	0.85
log-un.b	7.64	6.81	9.80	3.95
log-un.c	18.92	13.09	27.48	12.98

Figure 9: Simplification with COMPACT and 2-SIMPLIFY, Solution with REL-SAT.

formance of WALKSAT, a stochastic solver. As noted, stochastic solvers require tuning, and we tried to find the best parameters in each case. As Figure 10 shows, the results are mixed. On the *log* instances, we get 2-6 fold improvement, except for the log.d. In this instance, the simplified formula is solved faster, but simplification time is larger than solution time. However, on the *bw-dir* instances we get significant reduction in performance. If 2-SIMPLIFY ignores the transitive-reduction step (i.e., we maintain many binary clauses), we get somewhat different results (shown in the last column). These mixed result are not surprising as stochastic methods are know to be quite sensitive to the form of the formula.

5 Conclusion and Related Work

SAT instances with many binary clauses arise naturally in a number of important applications. The abundance of binary clauses in such problems can be exploited using 2-SAT solution methods and other specialized inference algorithms. Here, we presented 2-SIMPLIFY, a principled and efficient simplification algorithm that uses the transitive closure of the

Instance	WALKSAT	2S+WALKSAT	2S-TR+WALKSAT
bw-dir.a	0.06	0.20	0.16
bw-dir.b	4.13	79.55	7.97
bw-dir.c	39.26	—	17.79
bw-dir.d	94.47	—	160.61
log-dir.a	0.25	0.17	0.13
log-dir.b	0.45	0.0.26	0.17
log-dir.c	1.01	0.37	0.25
log.d	1.15	2.76	10.35
log-gp.a	8.20	1.20	4.24
log-gp.b	7.19	3.23	29.36
log-gp.c	20.34	7.95	72.26

Figure 10: Solution times using WALKSAT.

implication graph together with a novel implementation of hyper-resolution (i.e., the *derive shared literals* step) and *transitive reduction* to obtain a smaller equivalent formula. This leads to an approach that is faster, more powerful, and more efficient than the ad-hoc resolution of binary clauses used in [Brafman, 1999]. Our experiments on a set of encoded planning problems show that 2-SIMPLIFY is beneficial in conjunction with systematic solution algorithms: in virtually all tested experiments shorter solution times were obtained. In conjunction with a stochastic solver, the results were mixed, and the utility depends on the problem instance.

We are not the first to utilize binary resolution in this area. Larrabee used the implication graph to devise a SAT algorithm in the context of test-pattern generation [Larrabee, 1992]. Larrabee systematically generates satisfying assignments consistent with the implication graph. Any assignment that satisfies the non-binary clauses is a satisfying assignment for the whole formula. This method exploits the binary portion of the formula, but it does not utilize the power of contemporary variable ordering and search techniques.

2CL [Van Gelder and Tsuji, 1996] is a solver based on the Davis-Putnam-Logemann-Loveland algorithm [Davis *et al.*, 1962]. At each branch point, 2CL constructs the transitive-closure of the current implication graph and uses it to choose the next branching variable. Thus, 2CL is a dynamic extension of a key aspect of 2-SIMPLIFY. We have yet to experiment with 2CL. However, our initial attempt to produce a dynamic version of 2-SIMPLIFY along similar lines were not competitive with SATZ for two reasons: Maintaining an updated transitive closure is costly in terms of time and memory, and SATZ seems to obtain better information through its use of unit propagation, and faster. At this point, it seems that the techniques that 2-SIMPLIFY utilizes (i.e., implication graph analysis, restricted hyper-resolution, and transitive reduction) provide a basis for a good simplifier, but not necessarily a good systematic solver.

Acknowledgments: I am grateful to Yefim Dinitz for his help and advice on graph algorithms and for important comments on previous versions of this paper. This work was supported in part by the Paul Ivanier Center for Robotics and Production Management.

References

- [Aspvall *et al.*, 1979] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.
- [Bayardo and Schrag, 1997] R. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. AAAI-97*, pages 203–208, 1997.
- [Brafman, 1999] R. I. Brafman. Reachability, relevance, resolution, and the planning as satisfiability approach. In *IJCAI'99*, pages 976–981, 1999.
- [Cook, 1971] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*. ACM, 1971.
- [Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communication of the ACM*, 5(7):394–397, July 1962.
- [Ernst *et al.*, 1997] M. D. Ernst, T. D. Millstein, and D. S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1997.
- [Gomes *et al.*, 1998] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of 15th Nat. Conf. AI*, pages 431–437, 1998.
- [Kautz and Selman, 1996] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of the 13th National Conference on AI (AAAI'96)*, pages 1194–1201, 1996.
- [Kautz and Selman, 1999] H. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *Proc. 16th Intl. Joint Conf. on AI (IJCAI'99)*, pages 318–325, 1999.
- [Larrabee, 1992] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, pages 4–15, January 1992.
- [Li and Anbulagan, 1997] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. IJCAI-97*, 1997.
- [Robinson, 1965] J. A. Robinson. Automatic deduction with hyper-resolution. *Int. J. of Com. Math.*, 1:227–234, 1965.
- [Selman *et al.*, 1992] B. Selman, H. J. Levesque, and D. Mitchell. Gsat: A new method for solving hard satisfiability problems. In *Proc. of the 10th National Conf. on AI (AAAI '92)*, pages 440–446, 1992.
- [Selman *et al.*, 1994] Bart Selman, Henry A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. Nat. Conf. on AI*, pages 337–343, 1994.
- [Shtrichman, 2000] O. Shtrichman. Tuning sat checkers for bounded model checking. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification 2000*, 2000.
- [Van Gelder and Tsuji, 1996] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. American Mathematical Society, 1996.