Solving Non-clausal Formulas with DPLL search

Christian Thiffault[1], Fahiem Bacchus[1⋆], and Toby Walsh[2⋆⋆]

[1] Department of Computer Science, University of Toronto,
Toronto, Ontario, Canada
[cat|fbacchus]@cs.toronto.edu
[2] Cork Constraint Computation Center,
University College Cork, Ireland.
tw@4c.ucc.ie

1 Introduction

State of the art SAT solvers typically solve SAT theories encoded into CNF using DPLL based algorithms [1]. Most problems, however, are not originally expressed in CNF but contain arbitrary propositional formulae. The original problem must therefore be converted into CNF. Converting to a simple and uniform structure like CNF provides conceptual and implementational simplicity. Indeed, a number of key techniques that improve the effectiveness and efficiency of DPLL solvers exploit the simple clause structure. However, converting to CNF loses a considerable amount of information about the problem's structure. This is information that could be used to improve the search efficiency.

In this paper, we demonstrate that conversion to CNF is unnecessary. The techniques that are effective for reasoning with CNF theories can easily be adapted to work on unconverted Boolean circuits. We have implemented NoClause, a non-CNF DPLL like solver with similar raw efficiency to highly optimized clausal DPLL solvers. Furthermore, this solver can use the extra structural information still present in the unconverted circuit to improve its solving power. We present some simple techniques for taking advantage of this additional structural information, and show how a clausal solver without access to this structure can easily be mislead to perform expensive and unnecessary computation.

2 SAT solving using CNF

Many problems like hardware verification are naturally described using Boolean circuits. A Boolean circuit represents a propositional formula as a directed acyclic graph (DAG). Each node is either a Boolean operator, whose children are its operands, or a propositional symbol with no children. The DAG representation allow Boolean circuits to contain only one instance of each subformula. For example $A \Rightarrow (C \wedge D) \vee B \Rightarrow (C \wedge D)$ would be represented by a DAG with only one instance of the subformula $(C \wedge D)$. Boolean circuits are typically converted into CNF using linear Tseitin-sytle

⋆ Supported by the Canadian Government through its NSERC program.
⋆⋆ Supported by Science Foundation Ireland.

encodings [2]. A new propositional variable is introduced for each internal node in the DAG, and clauses added to equate the truth value of the variable and the subformula under the node. For example, in the above formula, we introduce the varible $B_1 \equiv C \wedge D$. Converting to clauses gives $(\neg B_1, C)$, $(\neg B_1, D)$, and $(\neg C, \neg D, B_1)$. Then the variables $B_2 \equiv A \Rightarrow B_1$ and $B_3 \equiv B \Rightarrow B_1$, which yield the clauses $(\neg B_2, \neg A, B_1)$, (A, B_2), $(\neg B_1, B_2)$, $(\neg B_3, \neg B, B_1)$, (B, B_3), and $(\neg B_1, B_3)$. Finally, the variable $B_4 \equiv B_2 \vee B_3$ is added with clauses $(\neg B_4, B_2, B_3)$, $(\neg B_2, B_4)$, and $(\neg B_3, B_4)$.[3]

Such an encoding is linear in the size of the original circuit. Nevertheless, such encoding has significant disadvantages. In particular, the encoding hides the structural information. The clauses no longer directly reflect the structure of the original circuit. For example, it is not immediately obvious that the B_i variables represent derived signals rather than input signals, that B_4 is upstream of B_1 in the original circuit, or that B_4 encodes an *or* gate while B_1 encodes an *and* gate. Of course in this simple example, much of that information can be gleaned from the clausal encoding. In general, whilst some of this information can easily be computed from the clausal encoding, some of it is not so easy. For example, it is intractable to determine which variables represent derived signals and which represent inputs to the circuit [4].

Another problem is that the CNF theory has more variables. The space of truth assignments from which a solution must be found has been enlarged by an exponential sized factor. This does not necessarily mean that the search for a solution is any harder in practice. It is the size of the explored search space that matters. Nevertheless, we shall demonstrate that the difficulty of searching this larger space is exacerbated by the lack of structural information.

Structural Information. A number of works show that structural information can help modern SAT solvers. For example, the Eqsat solver [5] offers significant gains by representing and exploiting biconditionals. Until very recently, it was the only solver able to complete the par32 family of problems which contain many biconditionals. More recently, the Lsat solver [6] has shown that using more extensive structural information can allow some problems, that are very hard for clausal solvers, to be solved quite easily. Theoretical results also show that structure can be used to derive branching decisions which provide exponential reductions in the size of the search space [7].

Branching on the Added Variables. Do we branch on the added variables or not? Some authors have suggested we should not, to ensure the search space does not increase [8, 9, 6]. However, there is complelling empirical and theoretical evidence to suggest that this is not desirable on all problems. The most robust SAT solvers do not limit their branching to the input variables. It is not difficult to show that restricting the solver to branching only on the input variables entails a reduction in the power of the proof system it implements. A number of results of this form have been given in [10]. These results show that there exist families of Boolean circuits on which a DPLL solver that branches only on the input variables (in the clausal encoding) will always explore an exponentially sized search tree (irrespective of how it chooses which variables it wants

[3] It is possible to build a more optimal encoding that only imposes the condition $B_4 \Rightarrow B_2 \vee B_3$ rather than equivalence as long as a node is not the descendant of an equivalence operator [3].

to branch on), while a DPLL solver that is allowed to branch on the derived variables can construct a constant sized refutation tree.

Theorem 1. *[10] There exists families of Boolean circuits such that a short resolution proof of unsatisfiablity exists if and only if branching on derived variables is allowed.*

Unfortunately, despite yielding exponential speedups on some problems, branching on derived variables can also give exponential slowdowns. Consider a formula of the form $PHP^n \vee (q \wedge p)$, where PHP^n is an unsatisfiable formula requiring an exponentially sized resolution refutation (e.g., the pigeon hole problem with n pigeons), and q and p are propositional variables. The clausal encoding of this formula contains the added variables $B_1 \equiv PHP^n$, $B_2 \equiv (q \wedge p)$, $B_3 \equiv (B_1 \vee B_2)$, and other variables added by the clausal encoding of PHP^n. If the solver first assigns $B_3 = \text{TRUE}$, then $B_2 = \text{TRUE}$ both q and p will be unit propagated to TRUE. This set of assignments satisfies the formula. However, the clausal theory will still contain the clauses encoding the subformula $B_1 \equiv PHP^n$ so the solver's job will not yet be completed. If the solver was then to set the input variables of PHP^n, any such setting would force a compatiable setting of B_1 and the solver would be finished. Similarly, if the solver was to set $B_1 = \text{FALSE}$ then it could find a setting of the variables in PHP^n that falsifies PHP^n and again it would be finished. However, if it made the wrong decision of first setting $B_1 = \text{TRUE}$, then it would be faced with having to produce an exponentially size refuation of PHP^n in order to backtrack to reset $B_1 = \text{FALSE}$. All of this work is unnecessary, but in the clausal encoding it is difficult to detect that the work is not needed.

How often such situations occur, and how much search is wasted by clausal solvers is an empirical question. Restarts would probably allow the solver to avoid spending exponential time on any single unnecessary refutation. Nevertheless, over the course of a large search a considerable amount of time might be expended in this kind of wasted effort. Our empirical evidence indicates that this can in fact be the case.

3 DPLL without conversions

Our technique for performing DPLL search without conversion to clausal form is quite simple. First we compress the DAG represenation of the circuit to ensure that it contains no duplicate sub-formulas. For example, we will have only one node in the circuit representing each propositional variable and that node will act as input to all of the gates the variable appears in (incluing not gates). Then we add a truth value field to each node. A satisfying assignment (solution) for a circuit is an assignment of truth values to all the nodes that labels the root TRUE and satisfies all of the gates, e.g., an *and* gate node must be labeled FALSE if any of its children is labeled FALSE, a *not* gate node must be labeled TRUE if its child is labeled FALSE, etc. To find a satisfying assignment, we assign the root node of the circuit to TRUE and propagate. We then perform a standard DPLL-like backtracking procedure. At every branch, we pick an unassigned node in the circuit, assign that node a truth value and recurse. If we fail to find a satisfying assignment, we assign the node the opposite truth value and try again. We ensure that the current partial labeling is always consistent with the gates by propagating truth values up and down the DAG. A contradiction is detected, and the search can backtrack, whenever a node has both of the labels TRUE and FALSE propagated to it. The same procedure can also test validity. If there is no an assignment of truth values to the nodes that labels the root FALSE and satisfies all of the gates then the circuit is valid, i.e. true in all assignments.

The propagation rules are fairly obvious and are based on the semantics of the gates. The same propagation rules are used in the Tableau based SAT solver BCSat [11]. For example, when a node is labeled FALSE, we propagate a FALSE label to all its parent that are *and* gates, and a TRUE label to all its parent that are *not* gates. Similarly, if we label an *or* node with FALSE, then we propagate a FALSE label to all of its children. One propagation rule that is not as obvious is when all but one of the children of an *or* node are labeled FALSE. In this case the label of the *or* node is equivalent to the label of the sole remaining unlabeled child. Hence, either propagate its label directly to the other. It is also possible to define propagation rules for more complex gates, like even and odd parity (the gate is true if its inputs have even/odd parity), exclusive or, equivalence, or even counting gates that require a particular number of their inputs to be true.

The nodes in the DAG are in one-to-one correspondance with the variables in the CNF encoding. The internal nodes correspond to the introduced variables, and the input nodes correspond to the original propositional variables in the circuit. Branching on the truth value of nodes is entirely analogous to branching on variables in the CNF encoding. The analogy with standard clausal DPLL goes even further.

Theorem 2. *If assigning a variable v the truth value x in the Tseitin CNF encoding of a circuit causes another variable v' to be assigned the truth value y by unit propagation, then assigning the node corresponding to v the value x will cause the node corresponding to v' to be assigned the value y by applying our propagation rules.*

Thus search on the circuit can duplicate the DPLL search on the clausal encoding. Reasoning with a circuit can, however, offer a number of advantages. First, we still have the structure of the problem explicit. Second, we can support much more complex inference like formula rewriting and propagation rules for more complex gates like counting gates. Third, we can propagate "don't care" truth values.

Don't care propagation The problem described earlier where a clausal solver might perform unnecessary work, can easily be addressed by using the circuit's structure. In particular, in addition to propagating TRUE/FALSE truth values through the circuit we can also propagate don't cares. If a child of an *and* node is labeled FALSE, then the *and* is FALSE and the values of all of the other children are *irrelevant*. Our procedure therefore also has a set of rules for propagating don't care values through the DAG. A node is labeled don't care when *none* of its parents depend on its value. Once a node is labeled don't care, we need not branch on it. We terminate search when all nodes have been labeled TRUE, FALSE, or don't care. In the previous example with $PHP^n \vee (q \wedge p)$, we immediately terminate once q and p are set to TRUE. All of the nodes in the PHP^n sub-circuit will become don't cares.

3.1 Using clausal solver techniques

A number of techniques have been developed recently to improve clausal DPLL solvers. One imporant technique is the use of *watch literals* to make unit propagation efficient.

Watch literal techniques are also used in our non-clausal solver to make propagation more efficient. A simple example is propagating TRUE to an *and* node. The *and* only becomes TRUE when all of its children are labeled TRUE. Rather than check the *and* node every time one of its children is newly labeled TRUE, we can designate one to be a "true-watch" child. Only when that child is labeled TRUE do we check the *and*. If the *and* has an unassigned or FALSE child, we move the watch to that child. Otherwise, all of the other children must be TRUE and we can now propagate TRUE to the *and*. A similar technique can be used to improve the efficiency of don't care propagation, single unassigned child propagation, etc.

Another important technique is clause learning, and using learned clauses to guide branching choices. In our non-clausal solver, a contradiction is detected when both TRUE and FALSE are propagated to the same node. We keep track of the propagation rules that caused labellings and the node assignments that fired these rules. We then backtrace to a set of node assignments whose conjunction entails the contradition. This contradictory set of assignments is converted into a clause and that clause stored in a clause database in exactly the same way as clausal DPLL solvers. We can even duplicate the 1-UIP clause learning technique used in the Zchaff solver. This store of learned clauses can then be used in the standard way. We can infer new assignments by unit propagation on the learned clauses, and we can use the clauses to compute a VSIDS branching heuristic.

We have implemented both of these techniques in our non-clausal solver, and report on the results in the next section. We can, however, use the circuit structure to make some improvements. In particular, we use the circuit structure to perform clause reduction on the shorter clauses. As in the Zchaff solver, we regard clauses of length with length less than 100 as "permanent" clauses. Longer clauses will be removed from the database whenever we reclaim space. With permanent clauses, we use the circuit structure to remove locally redundant literals. For example, suppose that n_1 is an *and* node and n_2 is one of its children, then we can remove n_1 from any clause containing n_2 without loosing any information. This corresponds to a neighbourhood resolution step: we implicitly have the clause $(\neg n_1, n_2)$ so from the clause (a, b, n_1, n_2, x) we can generate the subsuming clause (a, b, n_2, x). It is simple to identify the relations between nodes that are close together in the circuit. We can restrict our search for reductions by looking only for all parents and children that are implied by n. If one of those implied nodes is also in the clause we can remove n. We have found that such reductions provide a useful improvement in runtimes.

4 Empirical Results

We have implemented a non-clausal DPLL solver, NOCLAUSE using the ideas described above. We represent the input as a Boolean circuit, perform 1-UIP clause learning at failures, use Zchaff's VSIDS heuristic to guide branching, perform don't care propagation, and reduce on all learned clauses of size of 100 or less. We compared our results with the Zchaff solver. Zchaff is no longer the fastest SAT solver, but its source code is available. Hence, we were able to have better control over the differences between our solver and Zchaff. In particular, we duplicated as much as possible Zchaff's

	Zchaff		NoClause			
Problem	time	Dec./Sec.	Cls. Size	time	Dec./Sec.	Cls. Size
2pipe	0.13	48,938	35	0.28	17,429	20
2pipe_1	0.17	30,906	32	0.13	25,562	14
2pipe_2	0.24	27,767	38	0.30	18,990	20
3pipe	2.75	14,219	88	1.39	10,718	28
3pipe_1	2.42	10,719	87	7.60	5,245	53
3pipe_2	3.69	9,493	93	5.83	5,424	40
3pipe_3	6.79	7,924	105	6.81	5,471	57
4pipe	179.88	3,009	253	9.38	4,439	44
4pipe_1	25.63	5,120	158	34.18	3,350	82
4pipe_2	47.34	4,440	186	34.37	3,280	89
4pipe_3	139.29	2,818	254	58.85	2,874	112
4pipe_4	90.33	3,275	228	40.68	3,011	115
5pipe	51.92	6,448	258	31.81	3,209	96
5pipe_1	120.95	3,158	273	111.28	2,300	143
5pipe_2	132.48	3,001	276	169.01	2,147	168
5pipe_3	132.03	2,918	271	127.72	2,293	167
5pipe_4	833.67	1,672	406	270.03	1,863	210
5pipe_5	236.49	2,446	324	129.75	2,185	173
6pipe	4,550.92	1,150	619	282.26	1,544	235
6pipe_6	1,353.82	1,591	469	994.83	1,333	309
7pipe	12,717.00	978	900	1606.20	795	339
7pipe_bug	121.12	8,883	393	0.27	1,781	10

Table 1. Comparision of Zchaff and our NoClause solver on the fvp-unsat.2.0 test set. time: CPU sec. dec./sec: solver's raw speed in number of decisions made per seconds. Cls. Size: Average size of learned clauses.

branching heuristic and clause learning techniques so as to make the differences mainly dependent on our solver's use of the circuit structure. Although all of the techniques used in the most recent solvers are not fully known, we understand that they differ from Zchaff mainly in these two areas. If this is the case then those improved techniques for clause learning and branching could be imported into our solver, and we would expect a similar improvement in performance. Another factor in our experiments is that very few non-clausal test problems are available. Those that are have already suffered some loss of structural information by been encoded into ISCAS format which contains only *and*, *or*, and *not* gates. We expect to see even better performance on problems which have not been so transformed. All experiments were run on a 2.4GHz Pentium IV machine with 3GB of RAM.

Table 4 shows the runtimes of Zchaff and our solver NoClause on the fvp-unsat.2.0 test set (due to M. Velev). Zchaff took a total of 20,749 sec. to complete this test set where as NoClause required only 3923 sec. We also see that, although NoClause does not have quite the efficiency (measured in number of decisions the solver could make per second) as Zchaff, it is fairly close despite not using a more complex noncausal representation. Furthermore, as NoClause has not been extensively tuned, there does not seem to be any intrinsic reason why the DAG representation cannot be as fast as a clausal representation. Finally we see that generally speaking NoClause learns shorter clauses. We saw similar results on the other two test suites that we were able to obtain in both clausal and non-clausal form: fvp-unsat.1.0 and vliw-sat-1.1. On fvp-unsat.1.0 NoClause required 171.6 sec. to solve all 4 problems while Zchaff required 245 sec. On vliw-sat-1.1 NoClause required 983 sec. to solve all 100 problems, while Zchaff required 3472 sec.

We also tested the effectiveness of the two structure based techniques: don't care propagation and learned clause reduction. When we turned off don't care propagation, the runtime of NoClause on the fvp-unsat.2.0 test set jumped from 3923 seconds to 29,077 seconds—somewhat worse than Zchaff. With no learned clause reductions, the runtime of NoClause increased to 5470 sec. Thus it appears that don't care propagation is critical to performance, while clause reduction helps but is not as significant. These results were consistent with our experience on other problems not reported here.

[A bit more data on reductions]

5 Conclusion

Our results demonstrate that conversion to CNF is unnecessary. A DPLL like solver can reason with Boolean circuits just as easily as with a clausal theory. We have implemented NoClause, a non-CNF DPLL like solver with similar raw efficiency to highly optimized clausal DPLL solvers. Reasoning with Boolean circuits offers a number of advantages. For example, we can support much more complex inference like formula rewriting, as well as propagation rules for complex gates like counting gates. We can also use the circuit structure to simplify learned clauses, and to inform branching heuristics. NoClause is related to the tableau based non-CNF SAT solver BCSat [11] and the matrix based non-CNF SAT solver [12]. Indeed, it shares many of the same propagation rules with BCSat. A major difference with both these other non-CNF solvers is that NoClause also propagates "don't care" truth values. Such propagation appears to have a very significant impact on performance.

Our experimental results are very promising. We are often about to outperform a highly optimized solver like Zchaff. We expect that the results would be even more favourable if the benchmarks available to us had not already lost some of their structure. As we explained before, the ISCAS format only contains *and*, *or*, and *not* gates. There are many other ways in which we expect performance could be further improved. For example, more complex preprocessing of the input circuit, as in BCSat, is likely to offer major efficiency gains. More sophisticated structure based learning, branching, and non-chronological backtracking schemes are also likely to give significant improvements to performance.

References

1. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **4** (1962) 394–397

2. Tseitin, G.: On the complexity of proofs in poropositional logics. In Siekmann, J., Wrightson, G., eds.: Automation of Reasoning: Classical Papers in Computational Logic 1967–1970. Volume 2. Springer-Verlag (1983) Originally published 1970.
3. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation **2** (1986) 293–304
4. Lang, J., Marquis, P.: Complexity results for independence and definability in propositional logic. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning. (1998) 356–367
5. Li, C.M.: Integrating equivalence reasoning into davis-putnam procedure. In: Proceedings of the AAAI National Conference (AAAI). (2000) 291–296
6. Ostrowski, R., Grégoire, E., Mazure, B., Sais, L.: Recovering and exploiting structural knowledge from CNF formulas. In: Principles and Practice of Constraint Programming. Number 2470 in Lecture Notes in Computer Science, Springer-Verlag, New York (2002) 185–199
7. Beame, P., Kautz, H., Sabharwal, A.: Using problem structure for efficient clause learning. In: Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003). Number 2919 in Lecture Notes In Computer Science, Springer (2003)
8. Giunchiglia, E., Sebastiani, R.: Applying the Davis-Putnam procedure to non-clausal formulas. In: AI*IA 99: Advances in Artificial Intelligence: 6th Congress of the Italian Association for Artificial Intelligence. Volume 1792 of Lecture Notes in Computer Science., Springer (2000) 84–95
9. Giunchiglia, E., Maratea, M., Tacchella, A.: Dependent and independent variables for propositional satisfiability. In: Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA). Volume 2424 of Lecture Notes in Computer Science., Springer (2002) 23–26
10. Järvisalo, M., Junttila, T., Niemelä, I.: Unrestricted vs restricted cut in a tableau method for Boolean circuits. In: AI&M 2004, 8th International Symposium on Artificial Intelligence and Mathematics. (2004) Available on-line at http://rutcor.rutgers.edu/ amai/aimath04/.
11. Junttila, T., Niemelä, I.: Towards an efficient tableau method for boolean circuit satisfiability checking. In: Computational Logic - CL 2000; First International Conference. Volume 1861 of Lecture Notes in Computer Science., Springer (2000) 553–567
12. Otten, H.: A non-clausal Davis-Putnam proof procedure. In: Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI). (1997) 82