Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) 191-198

Research Note

Decomposing SAT Problems into Connected Components

Armin Biere armin.biere@jku.at
Carsten Sinz carsten.sinz@jku.at
Institute for Formal Models and Verification,
Johannes Kepler University, Linz, Austria

Abstract

Many SAT instances can be decomposed into connected components either initially after preprocessing or during the solution phase when new unit conflict clauses are learned. This observation allows components to be solved individually. We present a technique to handle components within a GRASP like SAT solver without requiring much change to the solver. Results obtained when applying our implementation in the SAT solver COMPSAT to a number of realistic examples show that components really do occur in practice. We also provide some evidence that component structure can be used to improve performance.

KEYWORDS: *SAT, connected components*

Submitted October 2005; revised January 2006; published February 2006

Introduction

It has often been observed that an analysis of the graph structure of a SAT problem may help to speed up SAT solving [1, 5, 13, 14]. The most basic type of structure is connectivity, where we view the variables of a formula in CNF as nodes of the graph and the clauses act as hyper edges. If a CNF can be decomposed into connected components according to this definition, then the components form independent SAT problems, and accordingly the whole CNF becomes satisfiable if and only if each component is satisfiable individually. In this paper we substantiate the hypothesis brought forward in [15, 16] that connected components really do occur in practice. We also give an argument why ignoring components may result in more work. Furthermore we present a simple technique, that takes advantage of connected components. It can easily be integrated with solvers based on a conflict driven assignment loop, such as GRASP [11] or CHAFF [12].

Commonly, SAT solvers do not use connectivity information. Thus the question arises, how SAT solvers perform if a problem has multiple connected components. Regarding solvers similar to CHAFF, the following observation can be made: Conflicts and learned clauses, as they are learned, are always contained within a single component. Furthermore VSIDS [12] and BERKMIN style decision heuristics [8] enforce decision variables that occur in recent conflict clauses. Therefore, as soon as a CHAFF like SAT solver *enters* a component and derives some conflicts within the component, it will *stay* there until it is shown to be unsatisfiable or becomes satisfied. In the first case, the whole search comes to an end and the only benefit of using information about connectivity could be in determining the component to enter first. In the second case, where a solution to the component is found,

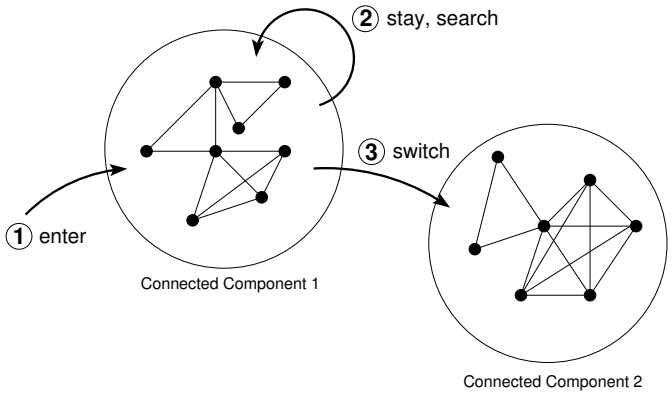

Figure 1. General working mode of a CHAFF like solver (without component detection) on a problem consisting of multiple connected components: (1) it enters a component, (2) it stays there until a satisfying assignment for the component is found (or it is shown to be unsatisfiable), and (3) it then switches to the next component. Learned clauses are always within one component. Backtracking on learned unit-clauses forgets solutions to already solved components, however.

the solver will *switch* to another component. This general schema is depicted in Fig. 1, where hyper edges are represented as cliques.

The only situation, in which component structure is ignored and may lead to more work occurs in the second case, the satisfiable case. Assume a unit clause is learned, which triggers backtracking and asserting the literal of the unit clause as new top level assignment. Let us call this event the start of a new *iteration*. If a new iteration starts and the unit involved is part of a certain component, then the standard implementation [11] will backtrack to the top level and erase all assignments, including those of already satisfied components. If the component in question becomes satisfied further on, then the solver has to reenter previously satisfied components again, since their satisfying assignment has been lost during non chronological backtracking at the start of the iteration.

As we will show in the experimental section below, *switching* really does occur in practice. Often unit clauses are learned, and many iterations happen. The latter effect also means that decomposing the CNF into connected components should be redone at the start of a new iteration, since assigning a variable on the top level has the potential of removing clauses, and can thus break components apart.

Poor Man's Version of Handling Connected Components

In order to take advantage of decomposing CNF into connected components, a general *proof engine* methodology as in [2] would be best. In this section we describe an alternative approach, which has been implemented in our solver COMPSAT. It does not incur much overhead and can easily be added to existing SAT solvers.

In principle, detection of connected components can be achieved by a simple graph traversal. Ordinary graph traversal, however, requires access to all edges resp. clauses in which a certain variable occurs. Since COMPSAT avoids full occurrence lists, as most

comparable SAT solvers do as well, and only watches two variables per clause, a simple graph traversal cannot be used, without increasing memory usage considerably.

To compute the decomposition, our implementation thus selects and stores a representative variable for each component. We then use the *union-find* algorithm [18] (which can be found in any textbook on algorithms) to determine component membership. In our implementation, each variable has an additional pointer to the representative variable of the component to which the variable belongs. Initially the pointer points to the variable itself. The decomposition algorithm then traverses the original clauses of the CNF, and, for each clause, merges the components of the clause's variables. During traversal, pointer chains to representatives are shortened recursively. In practice, generating component membership information with this algorithm exhibits an almost linear behavior in the number of original clauses as the analysis in [18] predicts. In a post decomposition step, after all representatives are known, the size of the components to which a variable belongs is calculated.

The CNF is decomposed initially after removing pure literals and removing all clauses satisfied by unit propagation. If a unit clause is learned and a new iteration starts, the SAT solver is forced to backtrack to decision level zero. After BCP of the new unit, without encountering a conflict, satisfied clauses can be removed. The resulting CNF is satisfiability equivalent to the original CNF and has less clauses which act as edges. Therefore the previous decomposition can further be refined using the same algorithm as for the initial and previous decompositions. Using an incremental algorithm would probably be faster.

When sorting literals for the next decision using VSIDS [12] the component structure is taken into account. Unassigned variables with small component size are preferred. Moreover, the search for still unassigned learned clauses in the BERKMIN decision heuristic [8] is stopped as soon as a learned clause outside of the current *active* component is found. The active component is defined by the last decision variable. When a variable of a new component different from the active component is picked as decision variable, a *switch* occurs and the active component is updated to the new component.[1]

If a switch occurs, the previous active component necessarily is satisfied, since all its variables are assigned without violating any clause within the component connecting them. Original clauses connecting variables in different components do not exist by definition and cross component learned clauses are removed by the garbage collector and are satisfied by any satisfying assignment of the original clauses anyhow. While switching the active component, the assignments to the variables of the previous active component are made permanent by setting their decision level to zero. The global decision level of the solver is not changed. This has the effect of remembering the assignments to these variables.

During backtracking variables with zero decision level are skipped and do not become unassigned. After the component is satisfied, a conflict that occurs later will never depend on decisions made within the satisfied component. Thus backtracking will always be non chronological and will jump over the satisfied component. There is also a *null component* represented by the invalid variable index zero, which is active initially and becomes again active after backtracking. Switching from the null component to a regular component is of course not counted as a regular switch.

1. Components are represented by their representative variable.

To summarize, the extensions that are needed to transform a CHAFF like solver with a VSIDS or BERKMIN decision heuristics into a solver that makes use of the component structure (as implemented in COMPSAT) consist of the following:

- Compute top-level components of the SAT instance (using the union-find algorithm).
- Remember assignments to variables in already solved components by setting the variables' decision levels to zero.
- Sort decision variables for the VSIDS heuristics according to the component structure.
- Iteratively refine the component structure on learned unit clauses (top-level assignments).

These extensions only add a constant overhead, except for the union-find algorithm. The union-find algorithm has to take (only original) clauses into account. In practice, its running time can be ignored compared to the time taken to run the garbage collector. In general, the garbage collector has the purpose to remove satisfied and inactive learned clauses. Using information about component structure, the garbage collector can remove cross component clauses immediately, which of course can only occur after unit clauses have been learned.

Experimental Results

We implemented the algorithm described in the previous section in our SAT solver COMPSAT [4]. In addition to the features described in [4], and besides handling connected components, the new version of COMPSAT uses a BERKMIN style decision heuristic [8] and implements garbage collection of learned clauses. Version 1.4 of COMPSAT used in the experiments is essentially the same as the version submitted to the SAT'05 SAT solver competition, except for some additional redundant code to gather and print extended statistics.

To determine empirically to which extent SAT-solvers can profit from the additional knowledge about a problem's component structure, we made two kinds of experiments: First, we used industrial SAT instances with a priori unknown component structure. And second, we artificially generated problems with a known number of components by appending several smaller SAT instances which were known to be satisfiable.

The first set of 115 benchmarks consists of all industrial benchmarks used in [19] after removing proprietary IBM benchmarks. The behavior of COMPSAT on these instances is very similar, whether component knowledge is used or not. Both variants were able to solve almost the same 64 benchmarks within a time limit of 1000 seconds and a memory limit of 1GB.[2] The version with decomposition using the techniques described in the previous section was able to solve one instance more.

We plot the running time for the 64 solved instances as scatter plot on the left of Fig. 2. The horizontal axis denotes the time taken by COMPSAT without decomposition, the vertical axis with decomposition. The data points slightly lie above the diagonal which is due to the additional time taken by the decomposition algorithm. On these 64 instances, the running time of the solver increases by 11%, calculated by the arithmetic mean of the

2. MiniSAT version 1.14 solves 84 instances within the same limits.

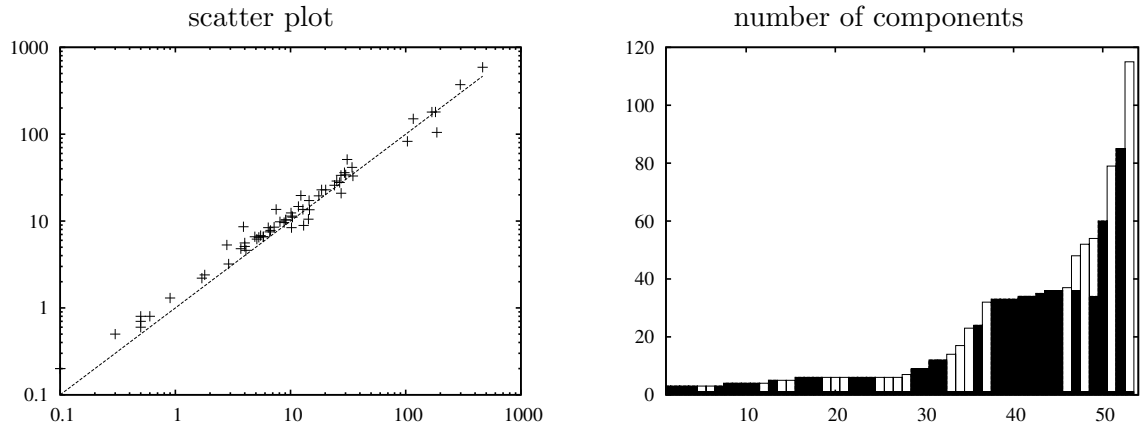

Figure 2. Industrial examples from [19]. The horizontal axis of the scatter plot on the left denotes the time in seconds taken by the original version of COMPSAT without using component structure and the vertical axis the time taken using decompositions. The figure on the right shows the maximal number of components as white bar resp. initial number of components as black bar on the vertical axis for each of the 54 out of 64 instances, for which at least one assignment to a variable could be saved. The bars representing the maximal and initial number of components for each instance are overlaid, with the white bar being always as tall as the black bar. The instances are sorted on the horizontal axis with respect to the maximal number of components found.

quotients of the running times, and by 20% by using the geometric mean. On the positive side, the implementation technique described in the previous section does not disturb the solver much. Unfortunately the plot also reveals that not much is gained either. This may actually be contributed to other inefficiencies in COMPSAT as discussed further down.

In 54 cases, out of the 64 instances solved by both versions of COMPSAT, assignments to variables could be saved: 1.56% saved assignments with respect to all variables on average and a maximum of 11.31%. On these 54 instances, 3025 regular switches occurred and 7597 iterations, which is equivalent to a total of 7597 learned unit clauses. However, note that learned units usually imply additional top level assignments. A total of 372310 top level assignments were implied by learned unit clauses for these 54 benchmarks not including 418360 top level assignments obtained by preprocessing the CNF with the pure literal rule and BCP. The number of initial components after preprocessing (black bar) and the maximum number of components (white bar) is shown on the right of Fig. 2. Note that trivial components, consisting of a single variable alone are not counted. These include all the saved and also all top level assigned variables. The effect of cross component learned clauses can be ignored, since only 3 of them were ever encountered for all 115 benchmarks. So our experiments corroborate the hypothesis that connected components and top level assignments really do occur in practice during the run of a CHAFF like solver.

As basic building blocks for the second kind of experiment we have chosen encodings of the factorization of integers. So, for a given integer x these instances ask, whether x can be decomposed into two integers p and q such that $p \cdot q = x$ holds. We have made experiments with several integers for which there is a unique solution. For a fixed value of x, we have then repeated these basic building blocks a number of n times and appended all these

instances (with shifted variable indices), resulting in a larger instance with n independent components. We then have run COMPSAT with and without component detection on these instances for varying values of n and measured the run-time. The results are shown in Fig. 3 on the left for $x = 2209$. In these figures, for comparison purposes, we have also included the run-times for MiniSAT [6] version 1.14.

It turned out that COMPSAT with component detection is clearly superior to COMPSAT without this detection. However, we did not observe a linear run-time behavior (in the number of components) for COMPSAT with component detection. Both versions almost uniformly reveal a quadratic behavior in the number of components, as can be seen from the auxiliary lines drawn in the figures (note that we are using log-log scaling in these plots, so polynomial functions are shown as straight lines with a slope proportional to the degree).

The similar asymptotic behavior of COMPSAT with and without component detection seems to be surprising. Initial profiling of COMPSAT running on these instances revealed that most of the time is spent sorting the variables according to their VSIDS score. Sorting variables is triggered quite frequently following the original ZCHAFF implementation and, moreover, in order to reflect changing component structure at the start of each iteration. This partially explains the quadratic behavior, which probably more likely is in $\mathcal{O}(n^2 \cdot \log n)$. As a consequence, the tested version of COMPSAT can not make full use of the component structure. There is no reason why the ideally achievable linear run-time behavior in n, the number of components, should not be realizable, in particular if a more advanced decision scheduler using a heap based priority queue as in MiniSAT version 1.14 is used.

Our experiments with a second set of basic satisfiable building blocks corroborates our observations. In these experiments, we have chosen hard satisfiable random 3-SAT instances with a clause-variable ratio near the phase transition point[3], and appended several of them to form a larger instance. Each individual component consists of 150 variables and 645 clauses, and we appended between one and a hundred of them to form the instances for our experiments. Fig. 3 on the right shows the results. Again, COMPSAT with component detection clearly beats COMPSAT without component detection. Our measurements also give a slight hint that component detection delivers a faster asymptotic behavior than the ordinary algorithm (as the slope indicates).

Related Work

Previous work on connected components for the SAT problem was mainly conducted in the context of model counting [3, 10, 14], although there are also approaches for constraint satisfaction problems in general [7, 9]. Bayardo and Pehousek extended the SAT solver Relsat 2.0 developed by Bayardo and Schrag in order to count models using connected components [10]. Their algorithm tries to detect components recursively at each node of a DPLL search tree. However, they do not take clause learning into account. Sang *et al.* carry on the recursive decomposition approach of Bayardo and Pehousek, and combine it with clause learning and component caching. Detection of components is accomplished by a simple depth-first search which is more expensive than the union-find algorithm we use, and may be too expensive for applying it to the SAT problem instead of the model counting

3. We used the Uniform Random 3-SAT instances from the SATLIB benchmark collection available at `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html`.

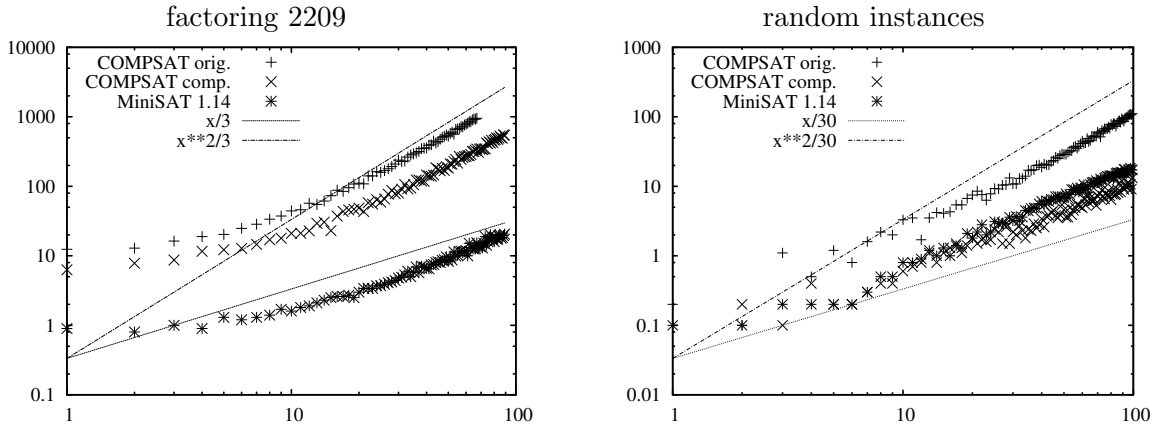

Figure 3. Run-times for SAT instances with known component structure. On the left, the factorization of 2209 was repeated between one and 100 times and concatenated. The number of repetitions is shown on the horizontal axis on a logarithmic scale. On the right, between one and 100 random 3-SAT instances were concatenated. Run times are shown on a logarithmic scale on the vertical axis. The solid lines are guides indicating linear and quadratic run-time behavior depending on the number of components.

problem. In contrast, our approach is much simpler and therefore much easier to implement. Especially, it becomes feasible to extend an already existing (CHAFF like) SAT solver with component detection. Moreover, our approach does not add too much computational overhead, so it will not slow-down solvers on instances where no component structure is present. Slater [17] compares different SAT solvers on clustered problem instances, without experimenting with specialized algorithms that exploit the structure, however.

Conclusion

We have shown that SAT solving can benefit from decomposing formulas in CNF into connected components. We presented a fast and easy to implement technique that takes advantage of component structure. Initial experiments are encouraging, and also revealed that connected components and top level assignments frequently occur in practice.

References

[1] F. A. Aloul, I. L. Markov, and K. A. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *Proc. 13th ACM Great Lakes Symp. on VLSI (GLSVLSI'03)*.

[2] G. Andersson, P. Bjesse, B. Cook, and Z. Hanna. A proof engine approach to solving combinational design automation problems. In *Proc. 39th Conf. on Design Autom. (DAC'02)*.

[3] P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. Memoization and DPLL: Formula caching proof systems. In *Proc. of the 18th Annual IEEE Conf. on Comput.*

Complexity (Complexity 2003).

[4] A. Biere. The evolution from Limmat to Nanosat. Technical Report 444, Dept. of Computer Science, ETH Zürich, 2004.

[5] P. Bjesse, J. Kukula, R. Damiano, T. Stanion, , and Y. Zhu. Diagnosis with tree decompositions. In *Proc. 6th Intl. Conf. on Theory and Appl. of Satisf. Testing (SAT'03).*

[6] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. 6th Intl. Conf. on Theory and Appl. of Satisfiability Testing (SAT'03).*

[7] E.C. Freuder and M.J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proc. of the 9th Intl. Joint Conf. on Artif. Intel. (IJCAI 1985).*

[8] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Proc. Design and Test Conf. Europe (DATE'02).*

[9] R.J. Bayardo Jr. and D.P. Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *Proc. of the 14th Intl. Joint Conf. on Artif. Intel. (IJCAI 1995).*

[10] R.J. Bayardo Jr. and J.D. Pehoushek. Counting models using connected components. In *Proc. of the 17th Nat. Conf. on Artif. Intel. (AAAI 2000).*

[11] J. P. Marques-Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD'96).*

[12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. 38th Design Automation Conference (DAC'01).*

[13] T. J. Park and A. Van Gelder. Partitioning methods for satisfiability testing on large formulas. *Information and Computation*, 162, 2000.

[14] T. Sang, F. Bacchus, P. Beame, H.A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of the 8th Intl. Conf. on Theory and Appl. of Satisfiability Testing (SAT 2004).*

[15] C. Sinz. Visualizing the internal structure of SAT instances. Extended version of an article in *Proc. of the 7th Intl. Conf. on Theory and Appl. of Satisfiability Testing (SAT 2004).*

[16] C. Sinz and E.-M. Dieringer. DPvis - a tool to visualize structured SAT instances. In *Proc. of the 8th Intl. Conf. on Theory and Appl. of Satisfiability Testing (SAT 2005).*

[17] A. Slater. *Investigations into Satisfiability Search.* PhD thesis, NICTA, Australian National University, Acton, Australia, 2003.

[18] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2), 1975.

[19] N. Éen and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. 8th Intl. Conf. on Theory and Appl. of Satisf. Testing (SAT'05).*