Decomposing Boolean formulas into connected components

T. Balyo

Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic.

Abstract. The aim of this contribution is to find a way how to improve efficiency of current state-of-the-art satisfiability solvers. The idea is to split a given instance of the problem into parts (connected components) which can be solved separately. For this purpose we define component trees and a related problem of finding optimal component trees. We describe how this approach can be combined with standard satisfiability solver decision heuristics to improve them. The proposed ideas were implemented and experimentally evaluated on a large set of benchmark problems. We provide results of these experiments.

Introduction

Boolean satisfiability (SAT) is one of the most important problems of computer science. SAT is well known in theoretical computer science since it was the first known example of an NP-complete problem [*Cook*, 1971]. SAT has many practical applications mainly in artificial intelligence. One of the first and most successful applications was solving automated planning problems via reduction to SAT [*Kautz, Selman*, 1992]. Other examples of applications are automated reasoning [*Robinson, Voronkov*, 2001] and hardware verification [*Velev, Bryant*, 2003]. Being NP-complete, SAT can not be solved in polynomial time unless P=NP. However this bound holds only for the worst case scenario. Many formulas which we need to solve for the applications of SAT can be solved in reasonable time using the current state-of-the-art SAT solvers.

One of the ways of improving efficiency of a SAT solving procedure is to divide the problem into smaller independent subproblems. The subproblems can be solved separately (in parallel). For problems with exponential time complexity this approach can help us to achieve exponential speedup. This idea has already been used to design efficient SAT solver decision heuristics [*Balyo, Surynek*, 2009;*Pipatsrisawat, Darwiche*, 2001] and also to improve satisfiability model counting ($\sharp$SAT) algorithms [*Bayardo, Pehousek*, 2000]. In this paper we will further investigate the possibilities of using connected components of SAT problems to enhance SAT solvers.

SAT Definitions

A Boolean variable is a variable with two possible values: $true$ and $false$. A literal is a Boolean variable or its negation. A clause is a disjunction (OR) of literals. A conjunctive normal form (CNF) formula is a conjunction (AND) of clauses. In the rest of the text by formula we always mean a CNF formula. A truth assignment ϕ for a formula F is a function $\phi : Vars(F) \to \{true, false\}$ which assigns truth values to each variable of F. Similarly a partial truth assignment assigns truth values to some of the variables of F. We say that a (partial) truth assignment ϕ satisfies a variable x if $\phi(x) = true$; a positive literal of the variable x if $\phi(x) = true$; a negative literal of x if $\phi(x) = false$; a clause if it satisfies any of its literals and a CNF formula if it satisfies all of its clauses. We say that a formula is satisfiable if there is a (partial) truth assignment that satisfies it. Satisfiability (SAT) is the problem of determining whether a given formula is satisfiable.

Solving SAT

There are several algorithms for SAT solving of various kinds, but the most successful ones are based on the Davis Putnam Logemann Loveland (DPLL) procedure[*Biere et. al.*, 2009]. DPLL is a depth first search of the space of partial truth assignments. We start with an empty partial truth assignment and try to extend it into a satisfying truth assignment. The search can be stopped if all the clauses are satisfied and we can immediately backtrack if there is a clause with all literals falsified by the current partial truth assignment. DPLL uses two additional enhancements: pure literal elimination and unit propagation. If a variable has only positive occurrences or only negative occurrences, then the literals of this variable are called pure literals. Such a variable can be immediately assigned to the proper value and make all its occurrences true. A clause is called unit if all but one of its literals are false and the remaining literal is unassigned. This literal has to be assigned to be true in order to satisfy the clause. This assignment can cause another clause to become unit and forces another assignment. The cascade of such assignments is called unit propagation. We present the pseudocode of DPLL as Algorithm 1.

Algorithm 1 DPLL($clauses, vars, assignment$) : $boolean$

```
if $\forall c \in clauses$ $assignment$ satisfies $c$ then
  return $true$
end if
if $\exists c \in clauses$ $assignment$ makes $c$ $false$ then
  return $false$
end if
$assignment = assignment \cup$ unitPropagation($clauses$, $assignment$)
$assignment = assignment \cup$ pureLiteralElimination($clauses$, $assignment$)
select $x$ such that $x \in vars \wedge assignment(x) = NULL$
return DPLL($clauses, vars \setminus \{x\}, assignments \cup \{x = true\}$)
      or DPLL($clauses, vars \setminus \{x\}, assignments \cup \{x = false\}$)
```

The performance of DPLL very much depends on the selection of decision variables. We use decision heuristics to select these variables. There are many very good decision heuristics already used by state-of-the-art SAT solvers. Our goal is to design a new one, which similarly to the divide and conquer principle will try to split the formula into parts and solve those independently. In order to exactly define this idea we will use a graph derived from a formula called an interaction graph [*Biere et. al.*, 2009].

Definition 1 *An interaction graph of the formula F is an undirected graph $G(V, E)$, where V is the set of variables of F and $(x, y) \in E$ if and only if there is a clause $c \in F$ that contains literals of x and y.*

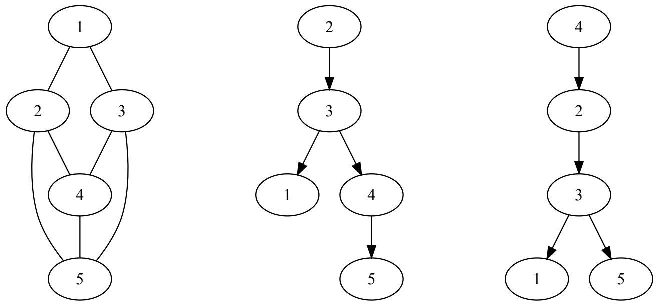

Figure 1. The interaction graph for the formula $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_4 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5 \vee x_3)$ and two of its component trees.

An example of a formula and its interaction graph is given on Figure 1 (left). We will be interested in the connected components of interaction graphs, because subformulas corresponding to different connected components can be solved separately since they have no common variables. If we consider the worst case time complexity of solving general SAT instances, then solving formulas by components can give us exponential speedup. For example if the interaction graph of a formula with n variables has two equally large connected components of $n/2$ vertices, then this formula can be solved in $2^{n/2} + 2^{n/2}$ time instead of 2^n, which is $2^{(n/2)-1}$ times faster.

Unfortunately, interaction graphs of SAT formulas rarely have more than just one connected component. However if we consider the DPLL algorithm and observe that extending a partial truth assignments is equivalent to removing vertices from the interaction graph, we can see that the interaction graph can get disconnected during the solving. To precisely describe this behavior we define the dynamic interaction graph.

Definition 2 *A dynamic interaction graph of the formula F and its partial truth assignment ϕ is an undirected graph $G(V,E)$, where V is the set of variables of F which have no assigned truth values by ϕ and $(x,y) \in E$ if and only if there is a clause $c \in F$ that contains literals of x and y.*

If G is the dynamic interaction graph for a formula F and its arbitrary partial truth assignment ϕ, then the subformulas corresponding to the connected components of G can be solved independently for any partial truth assignment that extends ϕ. We will use this property to design our decision heuristic. We want to assign those variables first, that will disconnect the interaction graph as quickly and as uniformly as possible. The remaining problem is to find those variables. For this purpose we need to define component trees and the component tree problem.

The component tree problem

Definition 3 *A $rootPath(v)$ of a vertex v in a rooted tree T is the set of vertices on the path from v to the root of T (including both v and the root). Let G_{-S} be a graph formed from G by removing the set of vertices S and all incident edges from G. A rooted tree T is a component tree for a connected graph G if G and T have the same vertices and for each vertex v that has at least 2 sons the following holds: The vertices in the subtrees of the sons of v are in different connected components of $G_{-rootPath(v)}$.*

An example of a graph and two of its component trees is on Figure 1. It is obvious from the given example that there can be several different component trees for a given graph. We need to compare different component trees, so we define the component value.

Definition 4 *The component value $C(v)$ of a vertex v in a component tree is defined as $C(v) = 1$ if v is a leaf and $C(v) = 2 \times \sum_{s \in sons(v)} C(s)$ otherwise. The component value of a tree is the component value of its root.*

The component values of the trees from the example on Figure 1 are 12 and 16 respectively. Component trees with lower component values will be preferred. We will call T an optimal component tree for a graph G if there is no other component tree for G that has lower component value than T. There can be many optimal component trees for a given graph. For example a clique on n vertices has $n!$ optimal component trees with 2^{n-1} being their component value.

The component tree problem is the problem of finding an optimal component tree for a given graph. The decision version is determining if there is a component tree of a given component value for a given graph. The decision version is clearly in NP, since the component tree itself is the certificate. It is unknown to the author of this paper whether it is NP-hard and thus NP-complete. However, if we do not require an optimal component tree, we can obtain a component tree easily by depth first search.

If we perform a depth first search (DFS) on a graph, then the tree edges of the DFS spanning tree form a component tree. Such a component tree can be very far from optimal. There are some examples of graphs, where this algorithm cannot find an optimal component tree no matter in what order we process the vertices [*Balyo*, 2010]. For these reasons we designed another algorithm called the component tree builder (CTB).

The pseudocode of CTB is given as Algorithm 2. It builds the component tree by connecting small component trees (initially consisting of only one vertex) into bigger ones. The $rootOf(v)$ method returns the root of the component tree to which v currently belongs. Every possible component tree can be constructed by this algorithm and thus also the optimal ones [*Balyo*, 2010]. The quality of the returned component tree depends on the order in which we process the vertices in the main loop. For the ordering of the vertices we will use the following greedy heuristics: select a vertex so that after its addition, the total increase of the component value is minimal. We experimentally compared CTB with the greedy heuristics and the DFS algorithm, where the next vertex is selected randomly. The results are presented on Figure 2.

The component trees often contain long segments of vertices with only one son. We call them linear segments. An exact definition follows.

Definition 5 *Let $L = x_1, x_2, \ldots, x_n$ be a sequence of vertices in a component tree T. If x_1 has no brother and $\forall i \in \{1, \ldots, n-1\}$ x_{i+1} is the only son of x_i then L is a linear segment of T.*

From the definition of the component tree we can easily prove that the order of vertices in the linear segments is not important. We can permute the vertices inside all the linear segments and we get a valid component tree with the same shape and thus with the same component value. This allows us to

Algorithm 2 The component tree builder algorithm

```
INPUT $G(V, E)$
$V' = \emptyset$, $E' = \emptyset$
for all $v \in V$ do
    $R = \emptyset$
    for all $s \in neighbor(G, v)$ do
        if $s \in V'$ then
            $R = R \cup \{rootOf(s)\}$
        end if
    end for
    $V' = V' \cup \{v\}$
    for all $r \in R$ do
        $E' = E' \cup \{(v \to r)\}$
    end for
end for
OUTPUT $G(V', E')$
```

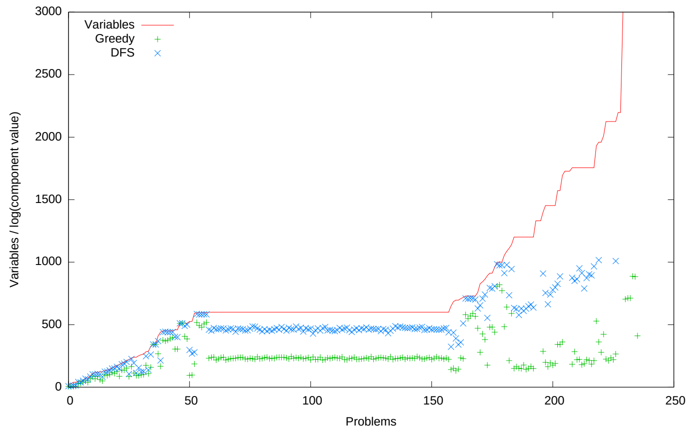

Figure 2. Comparison of the component tree construction algorithms on interaction graphs of SAT formulas. "Variables" denotes the number of variables of the formula, "Greedy" and "DFS" are the logarithms of the component values of the trees returned by the CTB algorithm using the greedy heuristics and the depth first search algorithm respectively.

look at the linear segments as sets of vertices and contract them into one vertex. The resulting tree is called a compressed component tree. An example of a component tree with its linear segments and its compressed component tree is given on Figure 3.

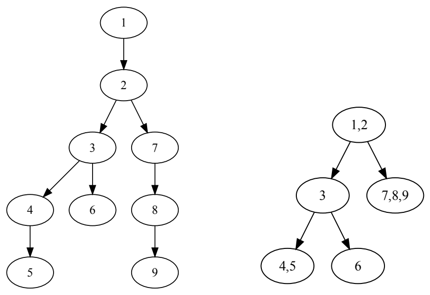

Figure 3. A component tree with linear segments $\{\{1, 2\}, \{3\}, \{4, 5\}, \{6\}, \{7, 8, 9\}\}$ and its compressed version.

SAT decision heuristics

Now we use the component tree to design decision heuristics for SAT solving. The algorithm is straightforward. We take the input formula and construct its interaction graph. We use our best available algorithm for component tree construction and find a component tree. Finally we contract its linear segments to get a compressed component tree. We will keep selecting decision variables from the root until all of them are assigned and the formula is disconnected. We proceed on the subtrees recursively to split the formula further. We will call this the component tree heuristics (CTH).

We have not yet specified the order of vertices in which we select them from the current compressed linear segment. If we pick them in a random order, the practical performance of the heuristics is very low. That is why we combine the component tree approach with state-of-the-art decision heuristics. These heuristics help us to order the variables within a linear segment. Now we will briefly describe some known decision heuristics and how they can be combined with CTH. Sometimes we will write "decision literal" instead of "decision variable", this means that we select the decision variable as well as the value that should be tried first. Some heuristics incorporate clause learning, which is described in [*Biere et. al.*, 2009].

Jeroslow-Wang (JW) [*Biere et. al.*, 2009] is a score-based decision heuristics. Each literal l has its score defined as $\sum_{cl\in F|l\in cl} 2^{-|cl|}$. The score is higher for literals with many occurrences in short clauses. An unassigned literal with the highest score is selected for assignment. The scores are computed once in the preprocessing phase. The combination with CTH is very simple, we select the literal with the highest score from our linear segment.

Dynamic Largest Individual Sum (DLIS) [*Biere et. al.*, 2009] is also score-based. The score of a literal is the number of not satisfied clauses containing it. This score is dynamic and thus needs to be recomputed when the partial truth assignment changes. The combination with CTH is analogous to JW.

Variable State Independent Decaying Sum (VSIDS) [*Biere et. al.*, 2009] is yet another score-based heuristic. For each literal l we define its score $s(l)$ and its occurrence count $r(l)$. At the beginning of solving, $s(l)$ is initialized to the number of clauses that contain l and $r(l)$ is set to 0 for each l. When a clause is added to the formula via clause learning, the occurrence counts of its literals are incremented by one. After each 255 decisions the scores of the literals are updated by the following formula: $s(l) = s(l)/2 + r(l)$ and $r(l)$ is set to 0 for each l. The literal with the highest score is selected for the decision. The combination with CTH is again analogous to JW (and DLIS).

BerkMin [*Biere et. al.*, 2009] also has literal scores based on the number of their occurrences. The decision literal is selected as the highest scored literal from the most recently learned currently not satisfied clause. The combination with CTH searches for the most recently learned unsatisfied clause that contains a variable from the current linear segment. Then we select the highest scored suitable literal from that clause.

Last Encountered Free Variable (LEFV) [*Balyo, Surynek*, 2009] defines no literal scores at all. It selects a literal from the clause which was the last not satisfied clause encountered during the most recent unit propagation. When combined with CTH, we select a literal from the last encountered clause that belongs to the current linear segment. If there is no such literal in the clause, we select a random literal from the linear segment.

Experiments

To measure the practical performance of our approach we implemented a SAT solver and all the above described heuristics. Our solver implements the conflict driven clause learning (CDCL) DPLL algorithm[*Biere et. al.*, 2009]. For clause learning we used the first UIP scheme[*Biere et. al.*, 2009]. Unit propagation was implemented using the 2-watched literals scheme[*Biere et. al.*, 2009] and our solver also incorporates restarting[*Biere et. al.*, 2009]. A detailed description of the solver is to be found in [*Balyo*, 2010].

We conducted experiments on various kinds of SAT benchmark problems. The formulas can be divided into two classes: random 3SAT formulas and structured formulas (modeling pseudo-practical problems). More exact description of the set of the used benchmark problems and their sources can be found in chapter 5 of [*Balyo*, 2010].

Selected results of our experiments are presented on Figure 4 for random formulas and on Figure 5 for structured ones. From the plots for random problems we can see that the performance difference is rather low for unsatisfiable formulas. Overall, there are many problems where the combined heuristics outperformed the original but in the majority of cases it did not.

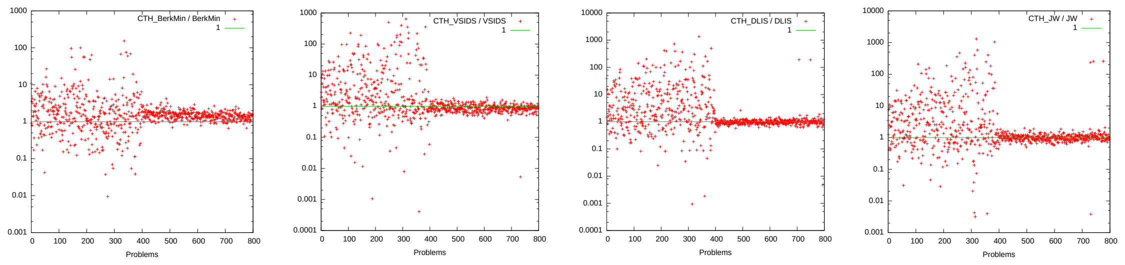

Figure 4. Comparison of the decision heuristics with their combined versions on random 3SAT formulas. The first 400 problems are satisfiable, the second half is unsatisfiable. If the cross is above the line then the combined version is weaker than the original.

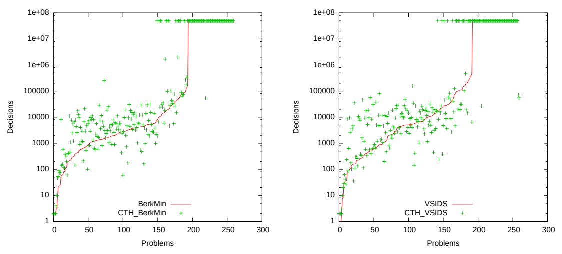

Figure 5. Comparison of the decision heuristics with their combined versions on random structured formulas. If the cross is above the line then the combined version is weaker than the original.

Conclusion

We defined a new general graph problem - the component tree problem and applied it to design a set of new decision heuristics for satisfiability solving via the DPLL algorithm. We experimentally compared the heuristics, but we did not manage to outperform the known heuristics used by state-of-the-art SAT solvers. Probably one of the reasons is that the component tree is a very rough approximation of the disconnection of Boolean formulas during DPLL. Component trees do not consider unit propagation, pure literal elimination or clause learning, which are important features of DPLL. We would like to remove these flaws in our future work.

Acknowledgments. I would like to thank my supervisor Roman Bartak and consultant Pavel Surynek for valuable discussions. This research is supported by the Science Foundation of the Charles University (grant No. 266111).

References

Balyo, T. and Surynek P., Efektivni heuristika pro SAT zalozena na znalosti komponent souvislosti grafu problemu, Proceedings of Znalosti 2009, 35-46, 2009

Balyo, T., Solving Boolean satisfiability problems, Diploma Thesis, Charles University in Prague, 2010

Bayardo, R. J. and Pehoushek J. D., Counting models using connected components, Proceedings of AAAI-00, 157-162, 2000

Biere, A. and Heule, M. and van Maaren, H. and Walsh, T. (editors), Handbook of Satisfiability, IOS Press, 2009

Cook, Stephen A., The complexity of theorem proving procedures, STOC, 151-158, 1971

Kautz, Henry A. and Selman, Bart, Planning as satisfiability, ECAI, 359-363, 1992

Pipatsrisawat K. and Darwiche A., A lightweight component caching scheme for satisfiability solvers, Lecture notes in computer science volume 4501, 294-299, Springer, 2007

Robinson, John Alan and Voronkov, Andrei, editors, Handbook of Automated reasoning (in 2 volumes), Elsevier and MIT Press, 2001

Velev, Miroslav N. and Bryant Randal E., Effective use of Boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors, Journal of Symbolic Computation, 35(2):73-106, 2003