Scalable Formula Decomposition for Propositional Satisfiability

Anthony Monnet
monnet.anthony_jean-luc@courrier.uqam.ca

Roger Villemaire
villemaire.roger@uqam.ca

Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal (QC) H3C 3P8, Canada

ABSTRACT

Propositional satisfiability solving, or SAT, is an important reasoning task arising in numerous applications, such as circuit design, formal verification, planning, scheduling or probabilistic reasoning. The depth-first search DPLL procedure is in practice the most efficient complete algorithm to date. Previous studies have shown the theoretical and experimental advantages of decomposing propositional formulas to guide the ordering of variable instantiation in DPLL. However, in practice, the computation of a tree decomposition may require a considerable amount of time and space on large formulas; existing decomposition tools are unable to handle most currently challenging SAT instances because of their size. In this paper, we introduce a simple, fast and scalable method to quickly produce tree decompositions of large SAT problems. We show experimentally the efficiency of orderings derived from these decompositions on the solving of challenging benchmarks.

Categories and Subject Descriptors

F.4.1. [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Logic and Constraint Programming*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Backtracking*; G.2.2 [**Discrete Mathematics**]: Graph Theory; F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous

General Terms

Algorithms, Performance, Experimentation, Theory

Keywords

SAT, propositional satisfiability, tree decomposition, scalability, DPLL

1. INTRODUCTION

The SAT problem consists in deciding if a given propositional formula is satisfiable, i.e. if there exists a truth assignment that makes the formula true. Furthermore, a satisfying assignment, or model, has to be returned if the formula is satisfiable. This problem has the particularity to be underlying in many reasoning systems. Indeed, the problems to be solved by these systems can often be expressed as a propositional formula, such that this formula is satisfiable iff the considered problem has a solution. Moreover, in the satisfiable case, a model of the formula can be translated as a solution of the original problem.

As the SAT problem is NP-complete [10], all known complete algorithms that solve it have a complexity exponential in the size of considered instances, including the widely used depth-first search algorithm DPLL [12]. Despite this obstacle, many efforts have been put during last decades to improve the effective efficiency of SAT solvers, mainly by improving the DPLL algorithm. Some of these improvements, such as conflict-directed backtracking and learning [30] or watched literals [33] decreased the time needed to solve most instances by several orders of magnitude wrt. a plain DPLL search. As a result, a wide choice of efficient, free and easy to use SAT solvers is available, and their performances are regularly compared during the SAT Competitions and SAT Races events [39].

Thanks to these off-the-shelf SAT solvers, solving a problem by encoding it as a SAT problem is often a convenient method. SAT has thus been used to express and solve problems from various areas, such as electronic design and verification [19, 43, 35], formal verification, including bounded model checking [7] and microprocessor verification [45], planning [26], scheduling [31], diagnosis [21], answer set programming [28], bioinformatics [29] and cryptanalysis [32, 17]. The performance of current SAT solvers allows most of these SAT-based systems to compete with or to outperform problem-specific strategies. Furthermore, any additional improvement in SAT solving will result in advances in all concerned domains. SAT is also used as a component for reasoning on more complex logics such as Quantified Boolean Formulas [6], SAT Modulo Theories [5], event calculus [34] and probabilistic theories [38]; as a consequence, progresses on SAT solving also reflect on the effective solving of the many problems expressible in these logics.

Real-world SAT instances derived from these application fields are typically very large; they commonly reach hundreds of thousands of variables and millions of clauses. Despite the tremendous progresses of SAT solvers, many of these large instances still remain challenging. A way to tackle this issue is to take advantage of the instance structure: in real-world instances, due to the semantic of the problems, variables generally directly interact with a limited number of other variables. They also tend to form clusters of highly interconnected variables, following the different parts of the problem description. As the structure of a propositional formula can be represented as a graph, structural graph theory tools can be applied to speed up SAT solving.

A particularly adapted concept is tree decomposition [37], which represents a graph by a collection of highly-connected subgraphs embedded in an arborescent metastructure. A tree decomposition is characterized by its width, the maximal size of its embedded subgraphs. For many graph problems that are NP-complete in general, one can design algorithms that, given a tree decomposition of the graph, solve the problem with a complexity that is exponential in the decomposition width, which may be significantly smaller than the size of the graph [3].

The main strategy for applying tree decompositions to SAT solving consists in representing the instance to solve as a graph, computing a tree decomposition of this graph, and using it to build a partial ordering of variables. This ordering is then used during a DPLL search. Besides the expected theoretical complexity result, several experimental studies [23, 27, 15] showed that variable orderings derived from tree decompositions can dramatically reduce the time needed to solve the satisfiability of a formula, compared with usual ordering heuristics implemented in state-of-the-art solvers. Unfortunately, the decomposition step of these implementations is not scalable enough to handle large real-world instances. Most of the time, the decomposition time alone is much longer than directly solving the instance through DPLL. Furthermore, the memory requirement often also gets prohibitive.

We therefore propose a simple and scalable method to produce tree decompositions through recursive separations, that we implemented to validate its efficiency in reducing SAT solving times. We focus on minimizing the time and space requirements of the decomposition, so we can decompose large instances without any significant time overhead. Our contribution is twofold: a heuristic allowing the practical decomposition of large propositional formulas, and an experimental evaluation showing that simple and rapidly-built decompositions can nevertheless significantly increase the efficiency of DPLL on these large instances.

In this paper, we will first set notions and notations on satisfiability solving and tree decompositions, before reviewing previous works using tree decompositions for the solving of SAT problems. We will then discuss the specificities of our SAT decomposition method, and report its experimental evaluation.

2. PROPOSITIONAL SATISFIABILITY AND TREE DECOMPOSITIONS

A propositional formula f on a set of variables $\mathcal{V}$ is recursively defined as a single variable $v \in \mathcal{V}$, the negation $\neg g$ of a formula g, or the connection of two formulas g and h by a binary operator amongst conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$). A formula is in *conjunctive normal form*, or *CNF,* iff it is expressed as a conjunction of disjunctions of literals, a literal being a variable or its negation. A disjunction of literals is called a *clause.* A CNF formula can be represented as a couple of sets $(\mathcal{V}, \mathcal{C})$, respectively the sets of variables and clauses it contains.

An *assignment* σ is a possibly partial function from variables to the set of boolean constants $\{true, false\}$. A variable $v \in \mathcal{V}$ is *instantiated* by σ if σ is defined on v. An assignment σ can be extended to formulas according to the usual semantic of operators. A total assignment σ is a *model* of f iff $\sigma(f) = true$. f is said *satisfiable* iff it has a model.

Given a propositional formula, the SAT problem consists in deciding if it is satisfiable, and returning a model if there exists one. Softwares designed to decide the satisfiability of propositional formulas are named *satisfiability solvers.* A solver is *complete* if it is always able to decide satisfiability on any formula, provided it is given the necessary time and memory resources. Most complete satisfiability solvers are based on the depth-first search DPLL algorithm [12]. DPLL basically tries to build a model of the formula by successively instantiating the problem variables. If the current partial instantiation violates a given clause (i.e. all variables of the clause are instantiated to the opposite polarity they appear with in this clause), the algorithms backtracks its search by undoing some instantiations and trying different instantiation polarities.

In practice, most satisfiability solvers only handle CNF formulas. Note that this requirement doesn't restrict their applicability, since any propositional formula has a logically equivalent CNF formula. Moreover, it is possible to transform any propositional formula f into an equisatisfiable CNF formula f' in linear time such that any model of f' restricted to the variables of f is also a model of f [44].

This normal form allows representing the structure of a formula as a graph, by characterizing the relationship between variables. Variables of a same clause are indeed tightly related, since all clauses of a formula must have at least one true literal for the formula to be satisfied. The most straightforward representation is a hypergraph where each variable is represented by a vertex and each clause by a hyperedge englobing all variables it contains. If we build the dual of this hypergraph, we obtain a representation of each clause by a vertex and each variable by a hyperedge on the range of clauses containing the variable. We will respectively call these representations the *primal* and *dual hypergraphs* of a CNF formula.

Tree decompositions are an algorithmic tool used to give an arborescent description of any graph or hypergraph by embedding it in a tree metastructure. Although they are originally defined on graphs [37], we will extend their definition to hypergraphs.

Given a hypergraph $\mathcal{H}(V, H)$, a tree decomposition $\mathcal{T}(N, E)$ of $\mathcal{H}$ associates to each of its nodes $n \in N$ a set $v(n) \subseteq V$ of hypergraph vertices, and a set $h(n) \subseteq H$ of hyperedges. The following constraints must hold:

- Every vertex and every hyperedge from $\mathcal{H}$ has to be included in at least one node from $\mathcal{T}$.
- If for some node $n \in N$ a hyperedge $h \in H$ is included in $h(n)$, then $h \subseteq v(n)$.
- If two distinct tree nodes n and n' contain a common vertex $v \in V$, then every tree node on the single path between n and n' must also contain v.

An important property of a tree decomposition is its width $w(\mathcal{T})$, defined by $w(\mathcal{T}) = \max_{n \in N} |v(n)| - 1$, i.e. the maximum number of hypergraph vertices associated to any tree node, minus one. The width of a hypergraph $w(\mathcal{H})$ can in turn be defined by the minimal treewidth amongst all its possible decompositions. The width of a hypergraph is correlated with its cyclicity: any tree has a width of 1, and on the opposite, any clique of n nodes has a width of $n - 1$. The treewidth can be seen as a measure of the effort needed to describe the hypergraph in an arborescent way, i.e. the amount of vertices that have to be grouped together in the same tree node.

For any pair of neighbour tree nodes n and n', the set of their common vertices $v(n) \cap v(n') = s(n, n')$ is called the *separator* between both nodes. Note that the term separator also denotes a set of vertices $S \subseteq V$ in a connected hypergraph $\mathcal{H}(V, H)$ such that removing these vertices from the hypergraph unconnects it. The terminology used is similar because every separator in a decomposition tree of a hypergraph $\mathcal{H}$ is also a separator in $\mathcal{H}$ itself.

Separation trees [11][1] are an alternative way of decomposing the structure of a hypergraph. They describe a recursive partitioning of hyperedges: the root contains all hyperedges of the hypergraph, each leaf contains exactly one hyperedge, and the hyperedge set of an internal node is partitioned amongst its children. Each internal node thereby defines a separator, as the set of variables its children have in common. Separation trees can be considered as a rooted and oriented version of tree decompositions; width-preserving transformations from one form to the other formalize this correspondence [11].

3. APPLYING TREE DECOMPOSITIONS TO SAT SOLVING

Tree decompositions are often used in order to decrease the complexity of (hyper)graph problems that are NP-complete in general, but polynomial if restricted to acyclic graphs. In this case, a subproblem is defined on each node n, generally the restriction of the original problem to the subgraph $\mathcal{H}_n(v(n), h(n))$. After all subproblems are solved, their solutions can be used to efficiently solve the global acyclic problem represented by the tree decomposition [3]. The exponential complexity is then limited by the size of the largest subproblem, which is the width of the decomposition. Tree decompositions can therefore be useful to efficiently solve instances with a limited treewidth, provided a low width decomposition tree is computed.

This kind of approach has been formalized in the case of SAT [1, 22], and is similar to *tree clustering* on constraint satisfaction problems [13]. A tree decomposition of the formula's primal hypergraph is computed. For each node of the decomposition, a subformula is defined as the conjunction of all clauses included in this node. Due to the properties of tree decompositions, the original formula is satisfiable iff there exists a set of solutions to the subproblems which are *compatible*, i.e. which set the same assignments to their common variables. In order to verify if such a set of solutions exists, a straightforward method is to search for all solutions of all subproblems, then to compute the natural join on these sets of solutions.

This method, that we will refer to as *explicit decomposition*, has indeed a time complexity that is exponential in the decomposition width. However, so is the space complexity, due to the enumeration and the memorization of all subproblems solutions. This is a serious drawback compared to usual depth-first search that only requires a linear amount of space, which strongly reflects on experimental performances [24, 41].

Thus an alternative way to use tree decompositions has been proposed [23, 9, 15, 27]. Instead of explicitly defining and solving subproblems, decompositions are only used to compute a variable ordering in the context of a DPLL search, that typically has a polynomial space complexity. Indeed, the order in which variables are instantiated has a major impact on the efficiency of DPLL, to find a satisfying assignment or prove unsatisfiability as fast as possible.

This strategy, that we will name *implicit decomposition,* uses tree decompositions to induce a partial order on variables. Decomposition nodes are statically or dynamically ordered, so that once a node is chosen, all of its variables must be instantiated before considering a different node, but the order of variables inside a node is left unspecified and can be dynamically decided by the usual solver heuristic. Moreover, the choice of the next node is usually restricted to neighbours of already instantiated nodes.

The main rationale behind implicit decompositions is that tree decompositions help capturing and exploiting structural informations from the instance. Indeed, it is guaranteed that two variables never occurring in the same decomposition node are not directly connected by a clause. Restricting the choice of variables inside a node therefore tends to instantiate successively variables that strongly interact with each other, which raises the chances to quickly produce propagations that help pruning the search space. Moreover, when the choice of the next decomposition node is restricted to the neighbourhood of previously treated nodes, all learned clauses are contained within a single node [9]. The size of learned clauses is thus limited to the decomposition width, which has a positive impact on solving performances [42].

Some works rely on separation trees and recursively instantiate the successive separators defined by their nodes, through a preorder traversal of the tree. This strategy adds the advantage of simulating a divide-and-conquer behaviour [23]: once a separator (and its ancestors in the tree) is instantiated, the remaining formula is disconnected along its subtrees, which thus define two mutually independent subproblems. Provided that the solver implements conflict-directed backtracking, if any of these subproblems is proved unsatisfiable under the current separator instantiation, the search will indeed directly backtrack to this separator, since the source of the conflict can't come from the other subproblem. This behaviour thus implicitly handles both subproblems separately. The complexity of DPLL being exponential in the size of treated problems, it is advantageous to solve disconnected subproblems independently, as it was shown experimentally [8]. Moreover, if the recursive division is balanced, the time complexity of implicit decomposition DPLL drops to $O(m \exp(s \log m))$, where m is the number of clauses and s the width of the separation tree [23].

The implicit decomposition scheme is therefore theoretically an interesting refinement of DPLL, since it reduces the time complexity of the solving without increasing the

[1]This concept is introduced as *dtree* in [11]; we will however name it separation tree to avoid any confusion with tree decompositions.

Algorithm 1 *Formula-Split*(V, H, v_k)

```
$V_\mathrm{m}^k \leftarrow \{v_k\}$
for all $h \in H$ do
  if $\exists\, v_1, v_2 \in h \mid v_1 < v_k < v_2$ then
    $V_\mathrm{m}^k \leftarrow V_\mathrm{m}^k \cup h$
  end if
end for
$V_\mathrm{l}^k \leftarrow \{v \in V \mid v < v_k\} \setminus V_\mathrm{m}^k$
$V_\mathrm{r}^k \leftarrow \{v \in V \mid v > v_k\} \setminus V_\mathrm{m}^k$
return $(V_\mathrm{l}^k, V_\mathrm{m}^k, V_\mathrm{r}^k)$
```

space complexity. Its relevance has also been confirmed experimentally: some implementations [23, 15, 27] managed to accelerate the solving of several benchmarks by introducing decomposition-driven orderings in state-of-the-art DPLL solvers.

However, the main drawback of decomposition methods is the computational cost of constructing a formula decomposition. As the width of the decomposition limits the time complexity of the subsequent DPLL search, this width should be minimized. Unfortunately, finding a minimal width decomposition of a graph is NP-complete [2], thus intractable. In practice, decompositions have to be built by heuristic methods, which allow building decompositions with relatively low time and space requirements. These heuristics don't provide any guarantee on the width of obtained decompositions, but use approximate strategies which are expected to produce relatively low width decompositions without requiring excessive time and space amounts.

For instance, some implicit decomposition implementations [23, 27] produce separation trees by recursively separating the dual hypergraph of the formula with the hypergraph partitioner hMETIS [25], which is itself a heuristic tool. The heuristic computation of hypertree decompositions, a generalization of tree decompositions [20], was also studied [14]. However, these experimental studies didn't consider instances of over 5,000 variables or 12,000 clauses. In contrast, the average size of real-world benchmarks from the 2009 SAT Competition is about 160,000 variables and 875,000 clauses. As demonstrated in Section 5, neither of these implementations can handle instances of this size.

4. FAST AND SCALABLE FORMULA DECOMPOSITION

As stated in the previous section, the scalability of the decomposition procedure is clearly the main obstacle to the use of implicit decompositions on large real-world satisfiability problems. We therefore propose a time- and space-efficient decomposition algorithm designed to handle large instances in negligible time. The proposed algorithm more exactly computes separation trees by recursively separating the formula. The main difference with previous separation tree implementations is the use of an extremely lightweight separation heuristic, in contrast with more complex and resource-consuming heuristics such as hMETIS.

This separation heuristic, *Formula-Split*, is described in Algorithm . It represents the formula to separate by the sets V and H of vertices and hyperedges of its primal hypergraph, and supposes that the set of variables/vertices is totally ordered: $V = (v_1, v_2, \ldots, v_{|V|})$ with $v_1 < v_2 < \ldots < v_{|V|}$. Given a split variable $v_k \in V$, *Formula-Split* partitions

Algorithm 2 *Separate-Node*$(n, \mathcal{T}(N, E))$

```
if $is\_separable(n)$ then
  $v_k \leftarrow (choose\_split\_variable(v(n), h(n))$
  $(V_\mathrm{l}^k, V_\mathrm{m}^k, V_\mathrm{r}^k) \leftarrow$ Formula-Split$(v(n), h(n), v_k)$
  if $V_\mathrm{l}^k \neq \emptyset$ and $V_\mathrm{r}^k \neq \emptyset$ then
    // $V_\mathrm{m}^k$ is a valid separator
    // $lc$ and $rc$ are the new left and right children of the
    node $n$
    // $lc$, $n$ and $rc$ correspond respectively to $V_\mathrm{l}^k$, $V_\mathrm{m}^k$ and
    $V_\mathrm{r}^k$
    $v(n) \leftarrow V_\mathrm{m}^k$
    $N \leftarrow N \cup \{lc, rc\}$
    $E \leftarrow E \cup \{\{n, lc\}, \{n, rc\}\}$
    $v(lc) \leftarrow V_\mathrm{l}^k$
    $h(lc) \leftarrow \{h \in h(n) \mid \exists v \in V_\mathrm{l}^k,\ v \in h\}$
    $v(rc) \leftarrow V_\mathrm{r}^k$
    $h(rc) \leftarrow \{h \in h(n) \mid \exists v \in V_\mathrm{r}^k,\ v \in h\}$
    // recursive separation of $lc$ and $rc$
    Separate-Node$(lc, \mathcal{T}(N, E))$
    Separate-Node$(rc, \mathcal{T}(N, E))$
  end if
end if
```

V in V_l^k, V_m^k and V_r^k (*left*, *middle* and *right* sets) as follows. Let V_m^k be the union of all hyperedges (clauses) that cross over v_k (having at least one variable strictly smaller and one variable strictly bigger) and of v_k itself. Let V_l^k and V_r^k be the sets of remaining variables respectively smaller and bigger than v_k. Then V_m^k separates V in two disconnected sets V_l^k and V_r^k, provided V_l^k and V_r^k are non-empty. Indeed, having two connected variables $v_\mathrm{l} \in V_\mathrm{l}^k$ and $v_\mathrm{r} \in V_\mathrm{r}^k$ would imply the existence of an hyperedge $h \in H$ such that $\{v_\mathrm{l}, v_\mathrm{r}\} \subseteq h$ with $v_\mathrm{l} < v_k < v_\mathrm{r}$. This would cause by construction $\{v_\mathrm{l}, v_\mathrm{r}\} \subseteq h \subseteq V_m^k$, in contradiction with the fact that $(V_\mathrm{m}^k, V_\mathrm{l}^k, V_\mathrm{r}^k)$ is a partition of V.

The computational cost of *Formula-Split* is minimal. Indeed, it only needs to find the smallest and biggest variable for every hyperedge, and to maintain the set of variables assigned to V_m^k. The complexity of the procedure is therefore linear wrt. the size of the formula $(O(|\mathcal{V}| + |\mathcal{C}|))$.

Algorithms 2 and 3 describe how *Formula-Split* can be recursively applied on formula subsets to produce a binary separation tree. The tree is initially reduced to a single node, containing all variables and clauses of the whole formula. The variables are partitioned by *Formula-Split*, and the node is split according to the obtained partition, provided it is really a separation of the current formula. This is the case when the side sets (the left and right set) are non-empty. The obtained side sets are in turn recursively separated, as long as they are considered *separable* by the predicate *is_separable*.

We can observe that, for every call to *Formula-Split*, the following properties hold:

- $|V_\mathrm{l}^k| + |V_\mathrm{r}^k| \leq |V|$
- $|V_\mathrm{l}^k| \leq |V| - 1$ and $|V_\mathrm{r}^k| \leq |V| - 1$

From these properties, it follows that the execution of *Separation-Tree(V, H)* will make at most $|V|$ calls to *Formula-Split*. Therefore, the complexity of *Separation-Tree* is quadratic $(O(|\mathcal{V}|^2 + |\mathcal{V}||\mathcal{C}|))$.

The algorithm as expressed is rather generic, since it depends of the following parameters:

Algorithm 3 *Separation-Tree*(V, H)

```
$N \leftarrow root$
$E \leftarrow \emptyset$
$v(root) \leftarrow V$
$h(root) \leftarrow H$
$Separate\text{-}Node(root, \mathcal{T}(N, E))$
**return** $\mathcal{T}(N, E)$
```

- the chosen ordering of variables in $\mathcal{V}$;
- the choice of the split variable v_k in the *Formula-Split* procedure;
- the *is_separable* predicate.

In the following experimental study, we chose these parameters as follows:

- As variables are numbered in a CNF benchmark description, this numbering is directly taken as the variable ordering in the *Formula-Split* procedure. The obvious advantage is that we avoid the computational cost of building a different ordering, which may result in a significant gain of time and space on large instances. Moreover, real-world instance encodings are generally organized according to the logical description of the problem, which means that two variables with close numberings are more likely connected. Using this ordering thus allows exploiting this structural knowledge from the encoding.
- For any call of *Formula-Split* on a set of ordered variables $\mathcal{V} = (v_1, v_2, \ldots, v_{|\mathcal{V}|})$, the split variable v_k is set to $k = |\mathcal{V}|/2$. This heuristic doesn't only avoid considering multiple split variable candidates, but also tends to balance the produced side sets, since in this case each of them can include up to $|\mathcal{V}|/2$ variables. Moreover, this upper bound on the size of side sets makes the complexity of *Separation-Tree* decrease to $O((|\mathcal{V}| + |\mathcal{C}|) \log |\mathcal{V}|)$.
- Finally, the predicate *is_separable* controls the depth of the produced separation tree. Preliminary results indicated that building decompositions of unlimited depth didn't benefit the search. Indeed, recursive separation progressively produces smaller variable sets. Due to the preorder traversal, a too deep recursion will result in an over-detailed ordering of a few variables near the end of search branches, which won't have any effect on the overall performances. At the opposite, it adds the overhead of handling a uselessly large tree. The heuristic limit we used is to separate nodes only if their variable set is larger than the first produced separator. It has the advantage to limit the separation tree depth without increasing its width.

5. EMPIRICAL EVALUATION

5.1 Experimental Settings

The goal of this section is to assert that the scalability of *Separation-Tree* allows decomposing large state-of-the-art SAT instances and, moreover, that using it in an implicit decomposition scheme helps improving the solver's performance. To that purpose, an implicit decomposition ordering based on *Separation-Tree* has been implemented in the state-of-the-art solver miniSAT 2.0 [16], which is regularly amongst the winners of SAT Competitions and Races[2].

In this implementation, the decomposition is explicitly represented as a binary tree. During the CNF parsing, a single node is built containing all variables. Leaves are then recursively split until the desired limit. The formula is decomposed after the preprocessing phase, which consists in miniSAT in a first round of unit propagations. This way we decompose a slightly simplified formula. Unlike the original miniSAT which uses a global priority queue to successively pick the decision variables, each node of the separation tree has its own priority queue containing its associated variables. This structure allows to restrict easily the variable decision to a given node of the separation tree.

The variable ordering strategy is similar to previous implicit decomposition implementations [23, 27], following a preorder traversal of the separation tree. When a variable choice occurs during the DPLL search, the next variable is picked in the current decision node from its priority queue, provided it isn't empty. Once all variables of a node are instantiated, the solver recursively instantiates all variables of its left subtree, then of its right subtree, before backtracking to continue the tree traversal.

As this static subtree choice may seem arbitrary, we tested several dynamic heuristics to choose subtrees according to the activity of their variables, a measure used by miniSAT to dynamically order variable decisions through the VSIDS heuristic [16, 33]; we however didn't obtain any significant improvement in performances, which rather slightly deteriorated due to the cost of these heuristics.

All following experimental results have been obtained on a 3.16 GHz Intel Core 2 Duo CPU with 3 GB of RAM, running a Ubuntu 9.4 OS. The benchmark series used in our tests were selected amongst application benchmarks from various SAT competitions [39]. They were respectively generated by the following applications:

- Rewriting termination problems generated by the termination prover AProVE [18];
- Network configuration problems generated by the model finder Alloy [36] ;
- Bounded model checking instances of liveness properties encoded as safety properties, generated by Cadence SMV [40];
- Verification of dereferenced pointers in the source code of the spam filter Dspam, generated by the software verification tool Calypso [4].

In all series, we discarded instances which could be directly solved by miniSAT in less than 2 seconds, since there is little interest in trying to reduce their solving time. The size of remaining instances ranges from 8,000 to over 2 million variables and from 29,000 to about 9 million clauses.

[2]This implementation is available at `http://www.info2.uqam.ca/~villemaire_r/Recherche/Minisat/100220minisat_decomp.tar.gz`

application	*instance*	*variables*	*clauses*	*sep. time*	*sep. width*
Termination Proving	AProVE09-06	77,262	263,137	0.04	8,216
	AProVE09-07	8,567	28,936	0.00	1,157
	AProVE09-15	94,663	305,105	0.04	10,541
	AProVE09-17	33,894	108,759	0.01	3,968
	AProVE09-20	33,054	108,377	0.01	4,296
	AProVE09-21	29,964	91,044	0.01	6,249
	AProVE09-24	61,164	209,228	0.03	4,354
Software Verification	dspam_dump_vc1080	118,298	375,379	0.09	6,150
	dspam_dump_vc1081	118,426	375,699	0.09	4,446
	dspam_dump_vc1093	106,720	337,439	0.05	5,352
	dspam_dump_vc1104	280,972	926,808	0.25	4,947
	dspam_dump_vc949	112,728	360,099	0.11	4,990
	dspam_dump_vc950	112,856	360,419	0.12	4,969
	dspam_dump_vc962	101,150	322,159	0.07	5,888
	dspam_dump_vc972	274,451	908,255	0.24	4,144
Bounded Model Checking	abp1-1-k31	14,809	48,483	0.01	1,590
	abp4-1-k31	14,809	48,483	0.01	1,590
	bc56-sensors-1-k391	561,371	1,778,987	0.16	123,461
	bc56-sensors-2-k592	850,398	2,694,319	out of space	
	bc57-sensors-1-k303	435,701	1,379,987	0.13	100,486
	dme-03-1-k247	261,352	773,077	0.12	29,530
	guidance-1-k56	98,746	307,346	0.06	6,454
	motors-stuck-1-k407	654,766	2,068,742	0.18	136,569
	motors-stuck-2-k314	505,536	1,596,837	0.14	104,245
	motors-stuck-2-k315	507,145	1,601,920	0.14	104,921
	valves-gates-1-k617	985,042	3,113,540	out of space	
Network Configuration	clauses-4	267,767	1,002,957	0.06	65,911
	clauses-6	683,996	2,623,082	0.14	176,973
	clauses-8	1,461,772	5,687,554	out of space	
	clauses-10	2,270,930	8,901,946	out of space	

Table 1: Experimental results of decomposition on selected SAT instances. *Instances* are grouped by the *application* they have been generated from. For each instance, the number of *variables* and *clauses* are listed. This table lists the *time* needed to generation a decomposition of the instance with the *Separation-Tree* procedure and the *width* of the obtained decomposition.

5.2 Decomposition scalability

Table 1 shows statistics of the decompositions performed by *Separation-Tree* on the selected benchmarks. For each instance are displayed the time needed to build the separation tree and its separation width, which is a lower bound on the treewidth of the corresponding decomposition tree.

Despite the considerable size of most instances, *Separation-Tree* was able to decompose all of them except the four largest. From these results, we can estimate that *Separation-Tree* can handle instances up to about 700,000 variables and 2.5 millions of clauses, which is over the average size of SAT 2009 application benchmarks. Moreover, the decomposition time always remains negligible, since it never exceeds a quarter of second. In most cases, the separation width obtained represents between 5% and 20% of the total number of variables, which is quite significant considering the simplicity of the heuristics used.

To compare the scalability of *Separation-Tree* with reference to other decomposition tools, we tested two publicly available implementations on the same set of benchmarks: the separation tree generator integrated in the implicit decomposition solver *Dtree-ZChaff* [23], based on a recursive use of hMETIS, and the software *htdecomp* [14] which implements various heuristics for building hypertree decompositions. *htdecomp* was used with the Bucket Elimination heuristic, which according to previous experimental evaluations was often the fastest implemented heuristic, hence the less resource-consuming [14].

Within our experimental configuration, *htdecomp* was only able to successfully decompose the three smallest selected benchmarks (AProVE09-07, abp1-1-k31 and abp4-1-k31, which all have less than 15,000 variables and 50,000 clauses). *DTree-ZChaff* also managed to decompose the same instances plus two others (AProVE09-17 and AProVE09-21, having less than 35,000 variables and 110,000 clauses). In all other cases, the implementations couldn't compute any decomposition due to a lack of memory space.

These experimental results clearly show that *Separation-Tree* is significantly more scalable than both other implementations, since it can handle instances over an order of magnitude larger. Furthermore, besides their limited scalability, *DTree-ZChaff* and *htdecomp* are also disadvantaged by their computation time. Indeed, the time needed by *htdecomp* to decompose the satisfiability instances represents at least 65% of the time needed to solve directly the instance with miniSAT. Moreover, in the case of *DTree-ZChaff*, the decomposition time is at least 150% of the direct solving time. *htdecomp* and *DTree-ZChaff* are therefore of little use in an implicit decomposition scheme, even on instances of moderate size, in contrast with the negligible decomposition time of *Separation-Tree*.

5.3 Implicit Decomposition Performances

After having confirmed the scalability of our decomposition heuristic, we will estimate the usefulness of the generated decompositions in an implicit decomposition scheme. Table 2 compares the solving time obtained on the benchmarks by both the original miniSAT implementation and our implicit decomposition modification, within a time limit of 4 hours.

Most termination proving instances are solved within a few seconds by both implementations, and are therefore not very relevant to compare their performances. However, the instance AProVE09-20 is considerably harder, and requires about 32 minutes to be solved by miniSAT. With the help of decomposition, however, the solving is almost twice faster (about 17 minutes).

The four network configuration instances are quite challenging due to their size (the largest one having about 2 million variables and 9 million clauses). If the first instance requires about half a minute in both cases, miniSAT can't solve the second problem within the 4 hours limit. Implicit decomposition solving only needs about 16 minutes to complete. The two last instances are unfortunately too large for our current implementation.

Bounded model checking problems are also solved faster in most cases: amongst the 8 instances solved by miniSAT, 6 are sped up by decomposition, up to a factor 3.5. We also manage to solve one instance on which miniSAT ran out of time.

Finally, the software verification benchmarks show the most contrasted behaviour. Half of the instances are sped up by implicit decompositions, sometimes dramatically: 2 are solved within 2 seconds but can't be solved by miniSAT within 4 hours. At the opposite, however, two relatively simple instances for miniSAT (20 seconds and 2 minutes respectively) run out of time with decompositions. These very contrasted results can certainly be explained by the particularly detailed decompositions obtained on this benchmark series (the variables are partitioned in more than 100 sets in average, against 14 sets in average on other benchmarks). The resulting orderings thus may have a stronger influence on the search.

These results indicate that implicit decompositions based on *Separation-Tree* can have a significant positive impact on the solving of benchmarks from various applicative fields.

6. CONCLUSION

In this paper, we described and implemented a scalable decomposition procedure designed to improve the DPLL solving of large SAT benchmarks. We showed that this procedure enhances significantly the scalability of tree decompositions compared with previous decomposition heuristics, and that using it in an implicit decomposition scheme can speed up the satisfiability solving of benchmarks from various applicative areas.

The extremely low time cost of our decomposition procedure leaves us room to consider the use of some slightly more complex heuristics, for instance on the choice of the initial variable ordering or of the split variable during a separation. This could allow trading off a little longer decomposition process against a lower decomposition width, which in turn could benefit the subsequent DPLL search, provided the scalability of the procedure isn't affected.

7. ACKNOWLEDGEMENT

We gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada on this research.

/bibliographystyleabbrv

8. REFERENCES

[1] E. Amir and S. McIlraith. Solving satisfiability using decomposition and the most constrained subproblem (preliminary report). In H. Krautz and B. Selman,

application	*instance*	*variables*	*clauses*	*status*	*org. time*	*dcmp. time*
Termination Proving	AProVE09-06	77,262	263,137	SAT	9.98	6.47
	AProVE09-07	8,567	28,936	SAT	3.80	1.16
	AProVE09-15	94,663	305,105	SAT	3.06	4.15
	AProVE09-17	33,894	108,759	SAT	12.77	27.87
	AProVE09-20	33,054	108,377	SAT	1,929.83	1,036.86
	AProVE09-21	29,964	91,044	SAT	2.17	0.59
	AProVE09-24	61,164	209,228	SAT	3.67	1.88
Software Verification	dspam_dump_vc1080	118,298	375,379	UNSAT	8.80	3.73
	dspam_dump_vc1081	118,426	375,699	UNSAT	time out	0.42
	dspam_dump_vc1093	106,720	337,439	UNSAT	127.21	1.51
	dspam_dump_vc1104	280,972	926,808	UNSAT	time out	time out
	dspam_dump_vc949	112,728	360,099	UNSAT	time out	time out
	dspam_dump_vc950	112,856	360,419	UNSAT	time out	1.24
	dspam_dump_vc962	101,150	322,159	UNSAT	115.22	time out
	dspam_dump_vc972	274,451	908,255	UNSAT	19.62	time out
Bounded Model Checking	abp1-1-k31	14,809	48,483	UNSAT	31.21	14.85
	abp4-1-k31	14,809	48,483	UNSAT	31.33	14.82
	bc56-sensors-1-k391	561,371	1,778,987	UNSAT	2,251.54	1,597.52
	bc56-sensors-2-k592	850,398	2,694,319	UNSAT	time out	out of space
	bc57-sensors-1-k303	435,701	1,379,987	UNSAT	time out	11,421.50
	dme-03-1-k247	261,352	773,077	UNSAT	out of space	time out
	guidance-1-k56	98,746	307,346	UNSAT	309.12	time out
	motors-stuck-1-k407	654,766	2,068,742	UNSAT	4,642.59	5,265.41
	motors-stuck-2-k314	505,536	1,596,837	UNSAT	680.19	243.28
	motors-stuck-2-k315	507,145	1,601,920	SAT	695.60	194.94
	valves-gates-1-k617	985,042	3,113,540	UNSAT	time out	out of space
Network Configuration	clauses-4	267,767	1,002,957	SAT	32.97	30.69
	clauses-6	683,996	2,623,082	SAT	time out	965.00
	clauses-8	1,461,772	5,687,554	SAT	time out	out of space
	clauses-10	2,270,930	8,901,946	UNSAT	61.71	out of space

Table 2: Experimental results of satisfiability deciding on selected SAT instances. *Instances* are grouped by the *application* they have been generated from. For each instance, the number of *variables* and *clauses* and the satisfiability *status* are listed. This table compares the running times of the original miniSAT implementation *(org. time)* and of the separation tree modification (*dcmp. time)*. In both cases, random decisions have been turned off, in order to have a more objective comparison. In some cases, solvers were unable to decide satisfiability within the given time (*time out*) or space *(out of space*) limits.

editors, *LICS 2001 – Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, pages 329–343. Elsevier, 2001.

[2] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2):277–284, Apr. 1987.

[3] S. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, Apr. 1989.

[4] D. Babić and A. J. Hu. Structural abstraction of software verification conditions. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference*, number 4590 in Lecture Notes in Computer Science, pages 366–378. Springer, 2007.

[5] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.

[6] A. Biere. Resolve and expand. In H. H. Hoos and D. G. Mitchell, editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004*, volume 3542 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2005.

[7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[8] A. Biere and C. Sinz. Decomposing SAT problems into connected components. *Journal On Satisfiability, Boolean Modeling and Computation*, 2:201–208, 2006.

[9] P. Bjesse, J. H. Kukula, R. F. Damiano, T. Stanion, and Y. Zhu. Guiding SAT diagnosis with tree decompositions. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing – 6th International Conference, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2004.

[10] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[11] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1–2):5–41, Feb. 2001.

[12] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[13] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, Apr. 1989.

[14] A. Dermaku, T. Ganzow, G. Gottlob, B. J. McMahan, N. Musliu, and M. Samer. Heuristic methods for hypertree decomposition. In A. F. Gelbukh and E. F. Morales, editors, *MICAI 2008: Advances in Artificial Intelligence*, volume 5317 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2008.

[15] V. Durairaj and P. Kalla. Exploiting hypergraph partitioning for efficient boolean satisfiability. In *Ninth IEEE International High-Level Design Validation and Test Workshop, 2004*, pages 141–146. IEEE Computer Society, 2004.

[16] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing – 6th International Conference, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[17] T. Eibach, E. Pilz, and G. Völkel. Attacking bivium using SAT solvers. In H. K. Büning and X. Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008, 11th International Conference*, volume 4996 of *Lecture Notes in Computer Science*, pages 63–76. Springer, 2008.

[18] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2 : Automatic termination proofs in the dependency pair framework. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 281–286. Springer, 2006.

[19] E. I. Goldberg, M. R. Prasad, and R. K. Brayton. Using SAT for combinational equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2001)*, pages 114–121. IEEE Computer Society, 2001.

[20] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[21] A. Grastien, Anbulagan, J. Rintanen, and E. Kelareva. Diagnosis of discrete-event systems using satisfiability algorithms. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 305–310. AAAI Press, 2007.

[22] M. Heule and O. Kullmann. Decomposing clause-sets: Integrating DLL algorithms, tree decompositions and hypergraph cuts for variable- and clause-based graph representations of CNF's. Technical Report CSR 2-2006, Swansea University Prifysgol Abertawe, 2006.

[23] J. Huang and A. Darwiche. A structure-based variable ordering heuristic for SAT. In G. Gottlob and T. Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1167–1172. Morgan Kaufmann, 2003.

[24] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1):43–75, 2003.

[25] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, Mar. 1999.

[26] H. A. Kautz and B. Selman. Planning as satisfiability. In B. Neumann, editor, *10th European Conference on Artificial Intelligence*, pages 359–363. Wiley, 1992.

[27] W. Li and P. van Beek. Guiding real-world SAT solving with dynamic hypergraph separator

decomposition. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 542–548. IEEE Computer Society, 2004.

[28] Z. Lin, Y. Zhang, and H. Hernandez. Fast SAT-based answet set solver. In *Proceedings of The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, pages 92–97. AAAI Press, 2006.

[29] I. Lynce and J. P. Marques-Silva. Efficient haplotype inference with boolean satisfiability. *International Journal on Artificial Intelligence Tools*, 17(2):355–387, Apr. 2008.

[30] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[31] S. O. Memik and F. Fallah. Accelerated SAT-based scheduling of control/data flow graphs. In *20th International Conference on Computer Design, VLSI in Computers and Processors*, pages 395–400. IEEE Computer Society, 2002.

[32] I. Mironov and L. Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In A. Biere and C. P. Gomes, editors, *Theory and Applications of Satisfiability Testing, 9th International Conference*, volume 4121 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2006.

[33] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535. ACM Press, 2001.

[34] E. T. Mueller. Event calculus reasoning through satisfiability. *Journal of Logic and Computation*, 14(5):703–730, Oct. 2004.

[35] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based detailed FPGA routing. In *12th International Conference on VLSI Design (VLSI Design 1999)*, pages 574–577. IEEE Computer Society, 1999.

[36] S. Narain. Network configuration management via model finding. In *Proceedings of the 19th Conference on Systems Administration (LISA 2005)*, pages 155–168. USENIX, 2005.

[37] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, Sept. 1986.

[38] E. Saad. Probabilistic reasoning by SAT solvers. In C. Sossai and G. Chemello, editors, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 10th European Conference*, volume 5590 of *Lecture Notes in Computer Science*, pages 663–675. Springer, 2009.

[39] The international SAT Competitions web page. `http://www.satcompetition.org`.

[40] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer*, 5(2–3):185–204, Mar. 2004.

[41] D. Singer and A. Monnet. JaCk-SAT: A new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, 7th International Conference (PPAM 2007)*, volume 4967 of *Lecture Notes in Computer Science*, pages 249–258. Springer, 2008.

[42] N. Sörensson and A. Biere. Minimizing learned clauses. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing, 12th International Conference*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

[43] P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, Sept. 1996.

[44] G. S. Tseitin. On the complexity of proofs in propositional logics. In J. H. Siekmann and G. Wrightson, editors, *Automation of reasoning, Classical papers on computational logic*, volume 2. Springer, 1983.

[45] M. N. Velev and R. E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, Feb. 2003.