Guiding SAT Diagnosis with Tree Decompositions

Per Bjesse, James Kukula, Robert Damiano, Ted Stanion, and Yunshan Zhu

Advanced Technology Group, Synopsys Inc.

Abstract. A tree decomposition of a hypergraph is a construction that captures the graph's topological structure. Every tree decomposition has an associated tree width, which can be viewed as a measure of how tree-like the original hypergraph is. Tree decomposition has proven to be a very useful theoretical vehicle for generating polynomial algorithms for subclasses of problems whose general solution is NP-complete. As a rule, this is done by designing the algorithms so that their runtime is bounded by some polynomial times a function of the tree width of a tree decomposition of the original problem. Problem instances that have bounded tree width can thus be solved by the resulting algorithms in polynomial time. A variety of methods are known for deciding satisfiability of Boolean formulas whose hypergraph representations have tree decompositions of small width. However, satisfiability methods based on tree decomposition has yet to make an large impact. In this paper, we report on our effort to learn whether the theoretical applicability of tree decomposition to SAT can be made to work in practice. We discuss how we generate tree decompositions, and how we make use of them to guide variable selection and conflict clause generation. We also present experimental results demonstrating that the method we propose can decrease the number of necessary decisions by one or more orders of magnitude.

1 Introduction

Tree decomposition [6] is a graph theoretic concept which abstractly captures topological structure in a variety of problems [4] such as constraint satisfaction [12], Gaussian elimination [21], database query processing [9], and image computation [13].

The topological structure of a Conjunctive Normal Form (CNF) formula can be represented as a hypergraph, where the vertices of the hypergraph correspond to the variables of the CNF and the hyperedges correspond to the clauses. Given a small treewidth tree decomposition for a hypergraph of a CNF formula, a variety of methods are known for deciding its satisfiability [12, 10, 3].

In this paper we report on our effort to learn whether satisfiability solving guided by tree decomposition can be made to work in practice. To do this, we attempt to find tree decompositions of small treewidth for significant problems, and to incorporate methods based on tree decomposition into a state-of-the-art SAT solver.

The end result of the work presented in this paper is a satisfiability checking method that given a bounded width tree decomposition of a problem instance will be guaranteed to run in quadratic time. We present the methods we use for generating tree decompositions and show how we make use of the tree decomposition to guide diagnosis and conflict clause generation. We also present experimental results that demonstrate that there are real-life SAT instances with small tree width, where our tree-sat method decreases the number of necessary decisions by one or more orders of magnitude.

2 Preliminaries

In the remainder of this paper, we will focus on augmenting GRASP-like [16] implementations of the Davis-Putnam-Loveland-Logemann (DPLL) method [11] with tree decomposition guidance. We refer readers unfamiliar with decision, deduction, and diagnosis components of such algorithms, including conflict graphs and backjumping, to [23].

3 Tree Decomposition

Given a hypergraph $G = (V, E)$, where V is a set of vertices and E a set of hyperedges with $e \subseteq V$ for each $e \in E$, a *tree decomposition* of G is a triple (N, F, χ) where

1. N is a set of nodes,
2. $F \subset N \times N$ is a set of arcs such that (N, F) forms an unrooted tree,
3. $\chi : N \rightarrow 2^V$ associates a subset of vertices with each tree node,
4. for every hyperedge $e \in E$, there is some node $n \in N$ such that $e \subseteq \chi(n)$,
5. for every $n_1, n_2, n_3 \in N$, if n_2 lies on the path in (N, F) between n_1 and n_3, then $\chi(n_1) \cap \chi(n_3) \subseteq \chi(n_2)$. This means that, for each vertex $v \in V$, the set of nodes that contain v form a subtree of (N, F).

Informally, this means that a tree decomposition of a CNF formula will be an unrooted tree, whose nodes contain subsets of the variables in the formula. This tree needs to fulfill the two properties that (1) the set of variables in each clause in the CNF needs to be a subset of some node, and (2) the set of nodes that contain a variable v from the original CNF must form a subtree of the tree decomposition.

The *treewidth* of a tree decomposition is $\max_{n \in N} |\chi(n)| - 1$. The treewidth of a hypergraph is the smallest treewidth of any of its possible tree decompositions.

Figure 1 shows an example of a CNF formula and an associated tree decomposition of treewidth 3.

4 Tree Decomposition and DPLL-SAT

We will now relate each of the core algorithmic components of modern DPLL-based SAT solvers—decision, deduction, and diagnosis—to tree decomposition.

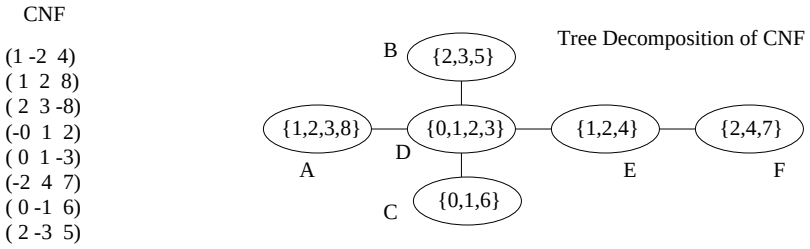

Fig. 1. A CNF formula and its tree decomposition

First, we show that a tree decomposition of a CNF formula will necessarily include paths corresponding to all possible chains of deduction resulting from assigning one or more variables. Next, we describe a set of conditions under which all chains from decision variables to an unsatisfied clause will pass through some common tree node, so that a legitimate conflict clause can be constructed from the variables in that node. Finally, we outline constraints on decision variable selection that insure these conditions are met.

Before we proceed further, however, we would like to note that a SAT procedure that restricts itself to constructing conflict clauses contained in tree nodes will be polynomial in the problem size for any class of problems with treewidth bounded by w. To see this, we can think of a SAT procedure as a series of conflict clause constructions. The number of conflict clauses constructed will be limited to $O(n3^w)$, where n is the size of the CNF formula, since any tree decomposition can be trivially reduced to have $O(n)$ nodes and each node with w variables can only generate 3^w clauses. When the i'th conflict clause is constructed, the CNF formula will have grown to size $O(n+i)$ from the addition of the prior conflict clauses. The time required to build the i'th conflict clause is linear in this size, since a chain of implications could propagate through a large fraction of the clauses. Thus the total SAT procedure will be bounded by

$$\sum_{i=0}^{n3^w}(n+i) = O(n^2 9^w)$$

There exists SAT algorithms for bounded treewidth problems that are linear rather than quadratic in the problem size [12]. However, these algorithms do not make use of the strength of DPLL solvers and as far as we know, none of these algorithms have managed to scale in practice. As our goal is to explore techniques that are effective for typical industrial problem instances, and DPLL solvers have proved to be very competitive in this context, our focus in this paper will thus be on modifying a DPLL solver to generate bounded conflict clauses even though the resulting complexity is superlinear.

4.1 Tree Decomposition and Deduction

Let us start by showing that for each chain of implications resulting from a variable assignment, there is a corresponding path in the tree decomposition.

The construction of DPLL solvers guarantees that every variable with an implied value is given that value because of some antecedent clause. Moreover, at the time of implication every variable in the clause except the implied variable must have been given some value, either through decision or through implication. Let us consider an arbitrary chain of implications $v_0, v_1, \ldots, v_k$, where v_0 is a decision variable, v_i was given an implied value before v_j whenever $0 < i < j$, and for each $i < k$, v_i appears in the antecedent c_{i+1} of v_{i+1}.

Lemma 1. *For any chain of implications there is path in the tree decomposition $n_0, n_1, \ldots, n_l$ together with a mapping $p : [0..l] \to [0..k]$, satisfying*

- *Either $n_i = n_{i+1}$, or n_i and n_{i+1} are adjacent in the tree decomposition.*
- $v_{p(i)} \in \chi(n_i)$
- $p(0) = 0$
- $p(l) = k$
- *If $p(i) = j$, then either $p(i+1) = j$ or $p(i+1) = j+1$.*
- *If $p(i) \neq p(i+1)$, then $n_i = n_{i+1}$.*

This node sequence is built up from paths in the tree between nodes containing successive antecedent clauses. To see this, focus on some particular subsequence v_i, v_{i+1}, $i > 0$, together with the antecedents c_i of v_i and c_{i+1} of v_{i+1}. By rule 4 of tree decomposition, clause c_i must be contained by some tree node n_{c_i}, and similarly clause c_{i+1} must be contained by some $n_{c_{i+1}}$ Variable v_i must be in clause c_i since c_i is the antecedent of v_i, and so v_i must be in node n_{c_i}. Variable v_i must also be in clause c_{i+1} since v_i, v_{i+1} is part of an implication chain, so v_i must also be in node $n_{c_{i+1}}$. Since variable $v+i$ is in both nodes n_{c_i} and $n_{c_{i+1}}$, by rule 5 of tree decomposition it must be in every tree node on the path between them. In this way we can build up the complete path in the tree decomposition by appending the paths joining each pair of successive antecendent clauses c_i and c_{i+1}.

As an example, consider the CNF formula in Figure 1. Assume that the variables v_1 and v_2 in the implication chain $7, 2, 8$ has the antecendent clauses $c_1 = (-2\ 4\ 7)$ and $c_2 = (1\ 2\ 8)$. The following is then a path and mapping corresponding to the chain:

i	0	1	2	3	4	5
n_i	F	F	E	D	A	A
$p(i)$	0	1	1	1	1	2
$v_{p(i)}$	7	2	2	2	2	8

4.2 Tree Decomposition and Diagnosis

Next we define a set of conditions under which a conflict clause can be constructed from variables within a single tree node.

Definition 1. *Given a CNF formula, a tree decomposition of it, and a partial assignment composed of decision and deduction assignments, a* core *subtree N_C is a nonempty maximal subtree that satisfies:*

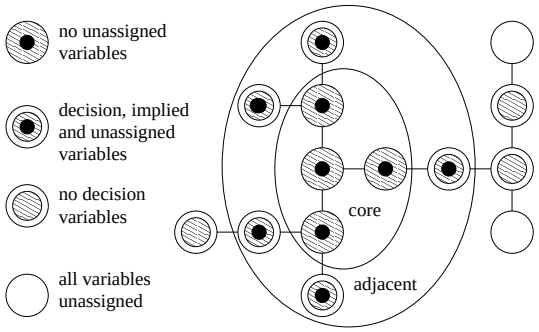

Fig. 2. Core and adjacent nodes

- *For each decision variable v in the partial assignment, some node in N_C or some node adjacent to N_C contains v.*
- *Every variable in every node of N_C is assigned.*

Definition 2. *Given a non-conflicting partial assignment, a further decision assignment is* admissible *relative to a tree decomposition if the decision variable is chosen from among the nodes adjacent to a core of the starting partial assignment.*

In Figure 2, the core subtree is made up from the four nodes in the middle that are fully assigned. Any variable that is contained in the five nodes adjacent to the core are admissible for assignment.

Suppose a admissible decision assignment to variable v results in a conflict. The unsatisfied clause cannot be contained in any of the nodes of the core N_C, because the starting partial assignment which was non-conflicting had already assigned values to all the variables in the nodes of N_C. $N \setminus N_C$ in general forms a forest of subtrees, one subtree of which contains n_D, the node adjacent to N_C which contains v. The unsatisfied clause must exist in this subtree, because no decision in n_D can cause any implication in any other subtree, since any implication path would have to traverse nodes in N_C but all variables in N_C have already been assigned. This leads to:

Lemma 2. *If a conflict is deduced from a admissible decision assignment, a conflict clause can be constructed from the variables in a single tree node, in particular from the variables in n_D.*

This can be seen by considering the chains of implications from the decision variables to the unsatisfied clause. Every implication chain to the unsatisfied clause starts at a node in $N_C \cup \{n_D\}$ and ends at a node containing the conflict in the subtree which contains n_D. N_C touches this subtree at n_D, therefore every implication chain from a decision variable to the unsatisfied clause must pass through n_D. A conflict clause must include enough variables assigned by decision or deduction to generate the conflict. Since every implication chain includes a variable in n_D, a conflict clause can be built from these variables.

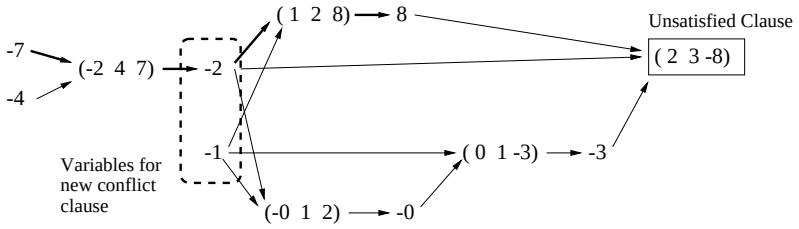

Fig. 3. An implication graph

As an example, consider the implication graph in Figure 3 where a conflict arises from the admissible decision to assign variable 1 the value 0 in the CNF from Figure 1. To see that this decision is admissible, note that earlier decisions has assigned variables 4 and 7 the value 0, which has led to an implied value for variable 2. Node F hence forms a core, which means that all variables in the adjacent node E are admissible. As the current conflict stems from an admissible decision, we can construct a conflict clause from the variables in a single node—the variables 1 and 2 in node E.

One concern regarding conflict clause construction is that the tree decomposition for the original CNF formula might not be a valid decomposition for the new formula that includes the constructed conflict clause. But since each constructed clause is contained within an original tree node, the original tree decomposition does remain valid as conflict clauses are added.

4.3 Tree Decomposition and Decision

A SAT procedure needs a way to select a decision variable whenever unassigned variables remain. We have seen that admissible decision assignments let us build conflict clauses from variables contained in a single tree node. However, admissible decision assignments are not available before a core subtree has been formed. We now describe an complete method to select decision variables that permits us to maintain our constraint on sets of variables in conflict clauses.

Definition 3. *Given a CNF formula and a partial assignment consisting of decision assignments and deductions, an unassigned variable is a* compatible *decision candidate relative to a tree decomposition of the CNF formula if it satisfies the following criteria:*

- *If no decision assignments have been made yet, then any unassigned variable is a compatible candidate.*
- *If no core subtree exists, then a compatible candidate must be contained in a node that contains all current decision variables.*
- *If a core subtree exists for the current partial assignment, a compatible candidate must be contained in a node adjacent to a core.*

That selecting decision variables from among compatible candidates is an effective strategy is established by the following two lemmas.

Lemma 3. *If each decision in a partial assignment is selected from candidates compatible with a tree decomposition, and unassigned variables remain, then a compatible candidate exists.*

The argument for this is inductive. As long as no core exists, the strategy calls for selecting variables from some common node. Once no unassigned variables are left in that common node, then that node becomes the seed for a core subtree, guaranteeing the existence of a core. A core is defined to be maximal, so nodes adjacent to a core must have unassigned variables. As long as one picks compatible candidates, further compatible candidates will exist until the assignment is complete.

Lemma 4. *If a decision assignment to a compatible candidate results in a conflict, then a conflict clause can be built from variables in a single tree node.*

To see this, we consider two cases. As long as a core does not exist, all decision variables come from some common node. A conflict clause can be constructed from the variables of this node, e.g. from the decision variables themselves. When a core exists, the compatible candidates are just those which give admissible assignments, so by Lemma 2 a conflict clause can be built from some single node.

With a decision variable selection strategy that supports conflict clause construction from single tree nodes, we can conclude that:

Theorem 1. *Given a tree decomposition for a CNF formula, decision and diagnosis can be performed so that for each conflict clause constructed, some node contains all the variables in the clause.*

5 Constructing Tree Decompositions

The effectiveness of the conflict clause construction method we have described relies on first constructing, for a given CNF instance, a tree decomposition of small width. Clearly some CNF instances will not have any small width decompositions, and for these the methods we have described will not provide a useful bound on the sizes of conflict clauses or their number. But we expect that many practical problems will have small width. For example, it has been observed [18] that digital circuits tend to have small cutwidth. Small cutwidth implies small treewidth, so we expect our method to be effective on a large fraction of digital circuits. Moreover, the reverse is not necessarily true, so our method has the potential of being effective even on some classes of problems that have an intractable cutwidth.

Finding a minimal width tree decomposition for an arbitrary graph is NP-complete [5], so an efficient general algorithm is unlikely to exist. For fixed k, checking if a graph has treewidth k and constructing a width k tree decomposition if it does can be done in time linear in the size of the graph [7]. Unfortunately, current algorithms grow in cost very rapidly with k, and are only practical for

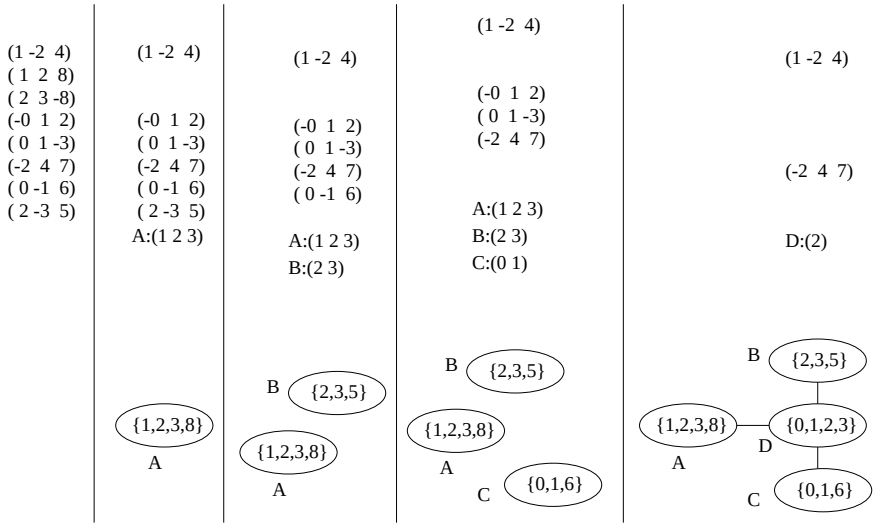

Fig. 4. Constructing a tree decomposition

very small k, roughly $k \leq 4$ [19]. Much more efficient algorithms have been developed for approximating k with bounded error [2], but even these appear to be too costly for industrial problems.

Due to the limitations of the direct approaches to computing a tree decomposition, we have taken a different approach. We rely on the facts that tree decompositions can be derived from orderings of hypergraph vertices [8]—CNF variables in our case—and that a plethora of robust and scalable CNF variable ordering methods is available. Given an ordering of variables, we use the following algorithm to compute a decomposition of a hypergraph representation G:

1. Let v be the next unprocessed variable in the variable order.
2. Add a new tree node n to the tree decomposition D with $\chi(n) = \bigcup_{e \ni v} e$
3. Update the hypergraph G by deleting all hyperedges containing v and adding a single hyperedge $e_n = \chi(n) \setminus \{v\}$.
4. Add arcs to the tree decomposition so that n is connected to every node n' whose hyperedge $e_{n'}$ just was deleted from G in step 3.
5. If unprocessed variables exists, goto 1, else we are done.

As an example, in Figure 4 we illustrate the generation of the tree decomposition from Figure 1 using the variable order $8, 5, 6, 0, 1, 3, 7, 2, 4$. The first variable in the order, 8, is contained in the clauses $(1\ 2\ 8)$ and $(2\ 3\ {-8})$, so when the node A is built, $\chi(A)$ is set to $\{1, 2, 3, 8\}$. The new hyperedge $e_A = (1\ 2\ 3)$ then replaces the two clauses with variable 8. Variable 5 generates $\chi(B) = \{2, 3, 5\}$ and $e_B = (2\ 3)$. Variable 6 generates $\chi(C) = \{0, 1, 6\}$ and $e_C = (0\ 1)$. The next variable in the order, 0, is contained in the clauses $({-0}\ 1\ 2)$, $(0\ 1\ {-3})$,

and $(0\ {-1}\ 6)$, as well as the added hyperedge e_C. So $\chi(D)$ will be $\{0,1,2,3\}$. Since node D incorporates hyperedge C, an arc is added between nodes C and D. Separate nodes could be built to reflect the elimination of variables 1 and 3, but since node D already includes all the variables involved, one can compact the tree by just letting node D serve for the elimination of all three variables $0, 1$, and 3. To reflect this, arcs are also added between node D and nodes A and B. The new hyperedge $e_D = (2)$ then replaces the clauses and earlier added hyperedges containing the eliminated variables. The creation of nodes E and F then continues in the same pattern.

The algorithm we use to build tree decompositions reduces the problem of finding a good tree decomposition to the problem of finding a good variable order. We have explored the use of two different methods for constructing variable orders for industrial-sized problems; a simple greedy method and a method based on linear arrangement. The simple greedy method we tried was the min-degree heuristic [2, 20] which is fast and known to give reasonable results. Each next variable in the order is determined on the fly from the reduced hypergraph from which earlier variables have been eliminated. The variable chosen to be next in the order is the variable for which $|\bigcup_{e \ni v} e|$ is smallest, being the size of the tree node to be constructed to eliminate the variable.

The second heuristic we explored is built using the MINCE linear placement engine [1]. The objects placed by MINCE are the hyperedges of the graph, corresponding to the CNF clauses. MINCE then generates a linear order of the clauses, attempting to reduce the cutwidth, the largest number of variables appearing in clauses on both sides of a cut. We then convert the clause order to a variable order by placing v_1 before v_2 if the last clause containing v_1 occurs before the last clause containing v_2. Since MINCE orders the clauses in a way to keep clauses with common variables near each other, our hope is that the tree decompositions generated from the MINCE clause order will have small width.

6 Implementing a Tree-based DPLL Solver

Let us consider the practicalities of integrating the tree decomposition approach to satisfiability solving into a modern Chaff-like [17] DPLL engine. In order to make use of a tree decomposition, we need to (1) modify the conflict clause generation; and (2) control the selection of decision variables so that we only use compatible candidates.

We solve the first problem by modifying the standard *1-UIP* [22] conflict clause generation slightly so that the conflict clauses that it returns are forced to contain variables exclusively from the last node a decision variable was selected from (the *choice node*)

In solving the second problem, we are free to use any variable order that respects the compability conditions. We will consider two ways of changing the Variable State Independent Decaying Sum (VSIDS) variable order [17].

The first of these approaches is the *static node* VSIDS order: Given a tree decomposition of a CNF problem, we generate a order on the nodes in the tree

Benchmark	Size (v/c)	Tree width	Static tree-sat # dec	Dyn. tree-sat # dec	Traditional sat # dec
dubois_50	150/400	4	**101**	**101**	2 647
dubois_500	1500/4000	4	**1 226**	2 002	58 316
dubois_1000	3000/8000	4	**3 351**	6 576	242 616
dubois_2000	6000/16000	4	**10 301**	18 223	712 153
addm_4_3	253/842	18	**938**	965	1 410
addm_4_4	433/1548	29	**2 653**	5 313	3 684
addm_4_5	661/2242	37	**6 141**	17 311	12 545
addm_4_6	937/3194	42	**23 735**	31 370	32 796
addm_4_7	1261/4314	51	1 480 277	**83 533**	134 426
addm_5_3	406/1367	29	**3 716**	12 060	8 344
addm_5_4	701/2382	41	**37 842**	64 132	42 486
addm_5_5	1076/3677	50	651 478	1 109 646	**166 847**
97686	4566/13987	170	**1914**	6228	9 485

Table 1. Experimental results

decomposition once and for all by computing the average initial variable score in each node. Each node's position in this order is static in the sense that it will not change during the search for a satisfying assignment. The largest node according to this measure is picked as the initial choice node in the tree (the *root node*). Whenever we need to select a new decision variable, we pick the best scored variable according to the VSIDS measure from current node. When no unassigned variables remains in our choice node, we move on to the best node adjacent to the core according to the static node order.

The second approach, the *dynamic node* variable order, differs from the static order in that we do not necessarily exhaust a choice node before we move on to the next node (with the exception of the initial root node, as this is required for compatibility). Instead, we pick a new choice node for each decision, by selecting that node adjacent to the core that will allow us to pick the highest scored variable according to VSIDS. In the dynamic node order, we pick the root node to be the node with the highest average of the smallest 10% of the nodes. The rationale for this is that we want to find a balance between being locked into a root node for the smallest number of decisions possible, and still make decisions using strong variables. In contrast to the static node order, we pick a new root node every time the proof search is restarted.

7 Experimental Results

In this section, we present the experimental performance of the tree-based SAT using the dynamic and the static variable order. Our objective is to show that satisfiability solving based on tree decomposition has potential in the sense that there are classes of formulas where it can be used to decrease the number of decisions significantly compared to a standard DPLL solver.

Our benchmark problems are a mixture of industrial and public benchmarks: The Dubois problems are a series of random benchmarks generated by the *gen-*

sathard program that is included in the DIMACS benchmark distribution [15]. The remaining problems are inhouse-generated equivalence checking problems. In particular, the *addm_x_y* examples are equivalence checks between different ways of implementing the addition of x y-bit numbers. The two implementations in each benchmark differ in the order they process individual bits from the different words. Note that the different *addm* benchmarks of varying size are not related in the sense that any one is a simple extension of the other—no adder trees in any of the problems have substructures that are even remotely similar.

We found that tree decompositions constructed using the MINCE-based heuristic generally gave significantly smaller treewidths than those constructed using the simple greedy heuristic, therefore the SAT results we report here used the MINCE-based heuristic. Since underlying MINCE engine, is randomized, different tree decompositions are generated for each run. We report the average value of ten SAT runs for the tree-based solvers.

To illustrate the different behavior of the two tree decomposition heuristics, we gathered information on the distribution of node sizes for each when run on the *addm_5_5* problem.

Percentile	Greedy	MINCE
20th	6	21
median	7	33
80th	12	41
max	95	46

The largest node in the MINCE-based decomposition, with 46 variables, is less than half that of the greedy decomposition. The greedy heuristic generates many small nodes and just a few large ones, while the MINCE-based heuristic generates a much more uniform distribution. Since the number of conflict clauses that can be constructed is exponential in the size of the nodes, the disadvantage of a few large nodes more than outweighs the advantage of many smaller ones. The computational cost of the MINCE-based heuristic is dominated by that of the underlying MINCE engine, making the greedy heuristic considerably faster. For example, on the *addm_5_5* problem the greedy heuristic was about 35x faster than the MINCE heuristic.

Our core DPLL solver is on par with Berkmin and ZChaff in terms of speed, but the additions for doing tree decomposition and handling the core are unoptimized. We therefore focus on comparing the proof methods in terms of the necessary number of decisions. This has the added benefit that it provides an implementation and platform independent measure of the potential of the methods.

As Table 1 indicates, the Dubois problems seem to have the interesting characteristic that their tree decompositions widths is held constant at 4 even when problem size increases. This means that they are easy for our tree-based solvers in the sense that very short conflict clauses will be needed to solve them. The experimental data confirms that both the static and dynamic tree-based DPLL solver needs orders of magnitude fewer decisions than our reference standard DPLL solver.

In contrast, the generated tree decompositions for the *addm_x_y* examples increase with the size and number of operands. The tree width for the larger examples ranges from 18 up to over 50. Still, the tree decomposition seems to be helpful, especially using the static variable order. However, for the very largest example (*addm_5_5*), the tree-based methods do many times worse than the plain solver. One of the potential reasons for this is that it becomes harder for our current tree decomposition engine to find a high quality decomposition as the problem size increases.

The industrial circuit 97686 has the largest tree width of all the benchmarks. Interestingly, it can still be solved using relatively few decisions by the static variable ordering algorithm. Even a tree decomposition with a high width can thus sometimes can be helpful.

As can be seen from the table, the static variable order seems to be better in almost all cases than the dynamic variable order. The results hence indicate that for these examples it is advantageous to keep the variables that are related by the nodes together in the variable order, rather than to try to emulate the VSIDS order as closely as possible.

Additional insight into the behavior of the conflict clause construction method we have described can be gained from the distribution of conflict clauses constructed. We gathered data for the *addm_5_5* problem:

Percentile	Static tree-sat	Traditional sat
20th	40	35
median	42	105
80th	44	141
max	45	351

This data shows that with conventional methods for decision and diagnosis, most conflict clauses constructed are longer than even the longest clauses constructed with our method based on tree decomposition.

We would like to note that although our results show consistent improvements in the number of decisions, we presently do not fare as well in terms of runtime. This is partly due to an unoptimized implementation of tree-based DPLL and partly due to the overhead of our unsophisticated tree decomposer. For example, the tree-based DPLL engines are a factor three to ten times slower per decision in terms of the pure DPLL engine on the *addm* examples, and we incur between three seconds and four minutes of overhead from generating the tree decompositions. However, some recent developments that we discuss in our conclusions in Section 9 suggest that a more efficient implementation of our method will have the potential to scale well also in terms of runtime.

8 Related Work

There have been other attempts to construct efficient decision methods for formulas with low treewidth. One such approach is Darwiche's Decomposable Negation Normal Form (DNNF) [10]. Formulas that have bounded treewidth can be

checked for satisfiability in linear time by first translating them into DNNF, and then applying a simple test. We expect there to be examples where satisfiability solving based on DNNF translation is superior to tree-based DPLL, since our theoretical complexity bound is quadratic. However, one appealing aspect of our approach is that the underlying SAT solving machinery that we are making use of is mature and is known to perform well in practice on industrial examples. Moreover, our light-weight integration makes it possible for us to interface to new solvers as soon as the state-of-the-art for DPLL solving advances.

The oracle we use for generating tree decompositions, MINCE, has previously been applied to the generation of variable orders both for BDDs and for SAT-solvers [1]. In this context, MINCE is used as a preprocessing step that generates an initial order. In contrast, we use MINCE to construct a tree decomposition that not only guides the variable ordering process in its entirety, but also guides the construction of conflict clauses.

There are strong parallels between our static node variable order and Amir and McIlrath's heuristic for generating a DPLL variable order from a decomposition of a propositional theory [3]. In this work the partitions are first ordered based on their *constrainedness*—the ratio of clauses contained in a given partition to the number of partition variables. The propositional variables are then ordered in a way that respects this partition order. A significant difference between our use of a given tree decomposition and Amir and McIlrath's is that we guarantee that all generated conflict clauses have length bounded by the treewidth of the decomposition.

9 Conclusions and Future Work

There has been a lot of research into tree decomposition, and there exists a rich theory about how tree decompositions can be used to solve NP-complete problems. However, the prior work in this field has not focused primarily on attempting to leverage tree decomposition to achieve speed ups on large real-life satisfiability problems. There has also been very little research that has aimed to combine the strengths of state-of-the-art satisfiability solvers with a tree decomposition generator that is practical for realistic problems.

The work that we have presented here represents a first step in this direction, and we hope that the results we have shown will stimulate further research. The simple approach we have presented already shows promise in the sense that it can decrease the number of necessary decisions for solving problems significantly, as witnessed by the order of magnitude improvements for the Dubois problems. Moreover, our work demonstrates that there exists heuristics that can process problems containing many thousands of variables and clauses in reasonable time, and still provide results that can help improve SAT solving efficiency. Finally, we have shown that there are structured problems from real-life, such as the *addm* problems, that have reasonable tree widths and where even an unrefined procedure that attempts to leverage tree decompositions can improve the decision count substantially.

We believe that tree decomposition can be a valuable tool for SAT. However, there is still work that remains to be done. For example on the benchmark problems, the best runs of MINCE often provide as few as half as many decisions than the average values that we have reported. As future work, we would therefore like to study how we can improve the tree decomposition engine, and tune our tree-based DPLL solver.

It is already clear that there is a lot more to be gained: In a parallel development to our original conference paper in SAT 2003, Huang and Darwiche have introduced a variable ordering heuristic that is a continuation of Darwiche's work on DNNF [14]. They guide decision variable selection in a DPLL solver using *DTrees*—the subclass of tree decompositions that correspond to full binary trees. Huang and Darwiche's variable selection heuristic uses the DTree to compute an order on the tree nodes that never changes during the execution of the search. However, just like in our case, the SAT engine is free to chose any decision variable within the current node until it is full. The most important difference between Darwiche and Huang's work, and the work presented in this paper is that we not only use the tree decomposition to guide decision, but use the structural information to enforce bounded conflict clause construction. Other, less significant, differences are that (1) we use a different oracle for generating tree decompositions, (2) we consider full tree decompositions rather than DTrees, and (3) our node order may change during the execution at the price of more runtime overhead. The DTree-based heuristic extends our experimental results by demonstrating that even in the case where conflict clauses are not bounded, runtimes can be improved significantly on a number of structured unsatisfiable benchmarks by navigating a subclass of tree decompositions in a completely static way. We are very excited about the results achieved by Huang and Darwiche, and we are eager to investigate how much further we can get by combining the benefits of their lower overhead approach to decision guiding with the power of our bounded conflict clause generation.

References

1. F. Aloul, I. Markov, and K. Sakallah. Faster SAT and Smaller BDDs via Common Function Structure. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 443–448, 2001.
2. E. Amir. Efficient Approximation for Triangulation of Minimum Treewidth. In *Proc. Conf. on Uncertainty in Artificial Intelligence*, 2001.
3. E. Amir and S. McIlraith. Solving satisfiability using decomposition and the most constrained subproblem. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, 2001.
4. S. Arnborg. Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposability - A Survey. *BIT*, 25:2–23, 1985.
5. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Algebraic and Discrete Methods*, (8), 1987.
6. H. Bodlaender. A Tourist Guide through Treewidth. *Acta Cybernetica*, 11, 1993.
7. H. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proc. ACM Symposium on the Theory of Computing*, 1993.

8. H. Bodlaender, J. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating Treewidth, Pathwidth, Frontsize, and Shortest Elimination Tree. *Journal of Algorithms*, 18:238–155, 1995.
9. C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *Proc. Int'l Conf. on Database Theory*, volume LNCS 1186, pages 56–70, 1997.
10. A. Darwiche. Compiling knowledge into decomposable negation normal form. In *Proc. Intl. Joint Conf. on Artificial Intelligence*, 1999.
11. M. Davis, G. Logeman, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(394–397), 1962.
12. R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–34, 1988.
13. A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik. Partition-Based Decision Heuristics for Image Computation using SAT and BDDs. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 286–292, 2001.
14. J. Huang and A. Darwiche. A structure-based variable ordering heuristic for SAT. In *Proc. Intl. Joint Conf. on Artificial Intelligence*, 2003.
15. D. Johnson and M. Trick, editors. *The Second DIMACS Implementation Challenge*. DIMACS series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1993. (see http://dimacs.rutgers.edu/challenges/).
16. J. P. Marques Silva and K. A. Sakallah. GRASP—a new search algorithm for satisfiability. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 220–227, 1996.
17. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT-solver. In *Proc. of the Design Automation Conf.*, 2001.
18. M. Prasad, P. Chong, and K. Keutzer. Why is ATPG easy? In *Proc. of the Design Automation Conf.*, 1999.
19. H. Roehrig. *Tree Decomposition: A Feasibility Study*. M.S. Thesis, Max-Planck-Instit. Inform. Saarbruecken, 1998.
20. D. Rose. Triangulated Graphs and the Elimination Process. *J. of Discrete Mathematics*, 7:317–322, 1974.
21. D. Rose and R. Tarjan. Algorithmic Aspects of Vertex Elimination on Directed Graphs. *SIAM J. Appl. Math.*, 34(1):176–197, 1978.
22. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. Intl. Conf. on Computer-Aided Design*, 2001.
23. L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. of the Computer Aided Verification Conf.*, 2002.