Greedy Clique Decomposition for Symbolic Satisfiability Solving

Yanyan Xu, Wei Chen, Kaile Su, Wenhui Zhang

Abstract

Motivated by the recent theoretical results regarding OBDD proof system, this paper applies a new variable grouping heuristic called greedy clique decomposition to symbolic satisfiability solving. Experimental results are compared against other state-of-the-art satisfiability solving tools, including Ebddres, Minisat, TTS and SSAT. We are able to show that with this new heuristic method, our implementation of an OBDD based satisfiability solver can perform better for certain instances, where conflict graphs possess a clique-like structure.

Keywords: OBDD Proof System, Symbolic Satisfiability Solving, Greedy Clique Decomposition, Propositional Proof System

1. Introduction and related work

The main motivation for the study of the propositional proof system lies in the connection between proof complexity and computational complexity [1]. However, in this paper we wish to explore the relatively less mentioned motivation: its application to practical satisfiability solving. Actually, each practical satisfiability solving procedure lays an implicit corresponding propositional proof system. For example, each DPLL [2] refutation of an unsatisfiable formula in CNF must contain a tree resolution refutation. Hence each DPLL refutation must have at least the same length of a tree resolution refutation. In other words, tree resolution can be viewed as the implicit propositional proof system inside the practical DPLL procedure. We can say that tree resolution roughly captures the power of DPLL in terms of refuting unsatisfiable CNF formulas.

For another example, the technique of clause learning and restarts is very successful in the recent development of complete satisfiability solvers. However, it is not until 2003 that Paul Beame et al. [3] obtained a theoretical characterization. They showed that DPLL with clause learning and unlimited restarts is equivalent to general resolution when refuting unsatisfiable formulas. Considering the fact that tree resolution is much weaker than regular resolution, which is in turn exponentially weaker than general resolution [4], we conclude immediately that DPLL with clause learning and unlimited restarts (This is a theoretical consideration, and in practice, the use of restart is rather limited.) is much more powerful compared to plain DPLL.

Generalizing the above two examples, for each practical satisfiability solving procedure, we can understand its power/efficiency better from studying the power of its corresponding propositional proof system. Given any propositional proof system $PS$, we can ask the following questions to try to deduce the power of its corresponding practical satisfiability solving procedure.

1). What is the relative efficiency of $PS$ compared to the more familiar resolution (general resolution, regular resolution and tree resolution) proof system? Is it exponentially stronger/weaker or incomparable?

After answering this question one can get an intuitive understanding of the power of this particular $PS$ and hence its corresponding practical satisfiability solving procedure.

2). Given any unsatisfiable formula $f$, is there any deterministic algorithm that returns the refutation of $f$ in $PS$ in time polynomial to the sum of the length of $f$ and the size of the shortest refutation of $f$ in $PS$?
In other words, given that a refutation of \( f \) exists, whether we can find it efficiently or not. This is the notion of automatization from [5]. This notion makes good sense for the relative weak propositional proof systems for which exponential lower bounds are known. The concept is relatively new, therefore there are few related results, and it is very hard to answer this question even for resolution or tree resolution [6] for the general case. This motivates us to ask this question with restrictions: for some specific subclasses of unsatisfiable formulas, is \( \textbf{PS} \) automatizable? In this case the \textit{subclasses} have to be carefully picked, and we expect practical experience to serve as the main intelligence for choosing these subclasses. If the answer to this question is yes, then we can get a practical procedure for efficiently refuting these particular subclasses of formulas.

The main point of explaining the above two questions is that rather than using theoretical results as merely a \textit{characterization}, we can also use them as a priori \textit{guide} when developing practical satisfiability solvers.

We show that this top-down approach (from theory to practice) can be beneficial by using results regarding OBDD proof system to motivate the development of practical symbolic satisfiability solving technique. More specifically, we first present the theoretical analysis [7] of the power of the OBDD proof system (the corresponding propositional proof system of symbolic satisfiability solving), showing that OBDD proof system is exponentially stronger than general resolution (Therefore, in some sense, symbolic satisfiability solving is even more powerful than DPLL with clause learning and unlimited restarts and this makes it much harder to automate). On the other hand, at least for pigeon hole principle \( \text{PHP}_{n+1} \), there exists a direct OBDD proof of polynomial size [8]. A natural next step is to try to generalize the way that we find the grouping of variables in pigeon hole case to an automatic mechanism, and apply it to other structural similar instances with the hope that we can get a polynomial time algorithm for these instances. In other words, we try to find out at least for some subclasses of unsatisfiable instances, whether OBDD proof system is automatizable. We get the inspiration from Pavel Surynek [9]. The technique of greedy clique decomposition is exactly what we need in this situation.

This paper explores the possibility of applying greedy clique decomposition as the variables grouping heuristics to symbolic satisfiability solving. Experiments show that this method is very fast for some typical instances which do possess a clique-like structural property in their conflict graph compared to some well-known satisfiability solvers, while our implementation can deal with both satisfiable and unsatisfiable instances and original method proposed by [9] can only decide very fast when the input is an unsatisfiable formula.

The paper is organized as follows: we present the definition and related results of the OBDD proof system in Section 2, which will provide the motivation for this work. The framework of symbolic satisfiability solving and how to apply the idea of clique decomposition as the grouping of variables heuristics are explained in Section 3. In Section 4 we present the experimental results and analysis. We conclude in Section 5.

2. OBDD proof system

We discuss two above-mentioned questions for the OBDD proof system. Explicitly considering OBDD as a propositional proof system is first stated in [7], in which the authors also provide definitions of four inference rules. For the formulation of the OBDD refutation rules, we adapt the idea from [10].

\textbf{Definition 1 (OBDD refutation).} An OBDD refutation of an unsatisfiable CNF formula with the variable set \( X \) is a sequence of OBDDs: \( C_1, C_2, C_3, \ldots, C_l \) such that \( C_1 \) is the OBDD for constant false and for each \( i \in \{1, 2, \ldots, l\} \) exactly one of the following conditions holds:

1. (Axiom Rule) \( C_i \) is the OBDD corresponding to one of the clauses;
2. (Join Rule) There exist \( j, k \) satisfying \( 1 \leq j \neq k < i \) such that \( C_i = C_j \land C_k \);
3. (Projection Rule) There exists \( k, 1 \leq k < i \) such that \( C_i = \exists x C_k \), \( x \in X \), and \( x \) does not appear in \( C_{i+1}, C_{i+2}, \ldots, C_l \);
4. (Weakening Rule) There exists \( k, 1 \leq k < i \) such that the solution space (space of satisfying assignments) of \( C_i \) is bigger than and strictly containing solution space of \( C_k \).

Greedy Clique Decomposition for Symbolic Satisfiability Solving
Yanyan Xu, Wei Chen, Kaile Su, Wenhui Zhang

175
We refer to $l$ as the length of the refutation proof, and the size of the refutation is defined to be $\sum_{i=1}^{l} \text{size}(C_i)$.

It is straightforward to see that this proof system is complete. In fact, first two rules alone are sufficient to make it complete. For an OBDD refutation system with first two rules, the refutation procedure for a CNF formula is straightforward: convert each clause to an OBDD using the axiom rule, then conjunct all these OBDDs together using the join rule. However, this may result in an exponential size explosion for intermediate OBDDs with respect to the size of the input formula, e.g., for the set of input formulas characterizing the pigeon hole principle $\text{PHP}^{n+1}_{n}$ [11]. For more comparison between this restricted OBDD proof system and resolution, we refer the readers to [12, 13]. The purpose of introducing the projection and weakening rules is to reduce the sizes of intermediate OBDDs. With the help of the weakening rule, it is proved in [7] that there exists polynomial size refutations for $\text{PHP}^{n+1}_{n}$, and in [8], we are able to show that with the projection rule alone, we can also do the same, without using the more powerful weakening rule.

It is worth pointing out that the weakening rule is not compatible with a complete SAT solving procedure. The reason is: in that case we need to deal with both satisfiable and unsatisfiable formulas. To weaken a formula $f$ is equivalent to disjunct it with another satisfiable formula $g$: $f \lor g$, and if $f$ is satisfiable then $f \lor g$ is satisfiable. Take the contraposition of the above statement, we get: if $f \lor g$ is unsatisfiable, then $f$ is unsatisfiable. This justifies the use of the weakening rule as an inference rule when refuting an unsatisfiable formula. However, $f \lor g$ is satisfiable does not necessarily imply that $f$ is satisfiable, therefore, the weakening rule is not justified when dealing with satisfiable formulas. The first three rules do not suffer from this deficiency. This is the reason why the term symbolic satisfiability solving [14, 15] corresponds to an OBDD proof system with the first three rules.

We present some basic results regarding the power of OBDD proof system and compare it with the more well-known resolution proof system.

**Theorem 1.** [7] OBDD proof system can polynomially simulate general resolution, moreover, it is exponentially stronger than general resolution.

The above theorem is established by first showing that OBDD proof system can polynomially simulate the resolution proof system, and then a concrete example is found for which the shortest proofs found by two different proof systems have exponential difference in size. This answers question 1. From this theorem, we conclude that in some sense symbolic satisfiability solving is even more powerful than DPLL with clause learning and unlimited restarts. This indicates that OBDD proof system is even less likely to be automatizable in general case than general resolution, since the more powerful a propositional proof system is, the less likely it is automatizable [6].

Next, we turn to question 2. Can we find some subclasses of unsatisfiable formulas and make OBDD proof system automatizable for these particular subclasses? Recall that in [8] we are able to give a polynomial size OBDD proof of the pigeon hole principle. We can try whether the variable grouping (or so-called bucketing) method we use in the pigeon hole case can be generalized and can be automatically decided and hence make OBDD proof system automatizable for pigeon hole problems and other structural similar instances (Some actual industrial benchmarks do have similar structure like $\text{PHP}^{n+1}_{n}$, and please refer to Section 4). We found that the method of greedy clique decomposition [9] is exactly what we need in this situation. We elaborate it in Section 3.

### 3. Decision procedure

First, we review the overall framework of symbolic satisfiability solving [14-16]. Given an unsatisfiable CNF formula, we first convert each of its clauses into corresponding OBDD, and then we

---

1. In fact, the projection rule is only a special case of the weakening rule.
2. From now on, when we speak of the OBDD proof system, we mean the OBDD proof system with the first three rules, unless we want to be specific.
3. The size of a resolution proof is the number of symbols contained in it.
divide its variable set \( X \) into several disjoint bucket sets: \( X_1, X_2, \ldots, X_k \), with each containing at least one variable. After that we join all the clauses in which the variables in \( X_i \) appear, and denote the OBDD we obtained by \( F(X_i) \) (Note that we follow the order \( F(X_1), F(X_2), \ldots, F(X_k) \) when constructing these \( F(X_i) \), and each clause is joined exactly once, i.e., if a clause is first joined in \( F(X_i) \) with \( k<i \), it won't be joined in \( F(X_i) \) even if variables in \( X_i \) do appear in this clause.), do the following:

\[
\exists X_1, \ldots, \exists X_k, (F(X_1) \wedge F(X_2) \wedge \ldots \wedge F(X_k)).
\]

That is, we eliminate (using existential quantification) all variables in each bucket at a time.

In this case, how we divide \( X \) into these buckets and the relative order these buckets are quantified out plays an important role affecting the sizes of the intermediate OBDDs. The existing implementation and tools known to us \([14-16]\) treat each variable as a single bucket and no efforts are made to discover the more natural variable grouping of the input formula. In this paper we provide a method to discover effective variable groupings automatically and utilize this information to improve the performance of symbolic satisfiability solving, at least for some particular subclasses of input formulas. The greedy clique decomposition proposed by \([9]\) may suit this purpose well (Note that in this case, we treat each clique as a bucket.).

### 3.1 Greedy clique decomposition

A clique is a complete subgraph. The greedy clique decomposition decomposes the conflict graph of the input formula into different cliques using a simple greedy method.

**Algorithm 1: Greedy Clique Decomposition**

```plaintext
1 GCD(G)
2 Input: the conflict graph \( G = (V, E) \)
3 Output: clique[clique_num]
4 begin
5   clique_num = 0;
6   find the vertex \( v \) with the largest degree in graph \( G \);
7   while \( (n \neq 0) \) do
8     add \( v \) and all its neighbors to clique[clique_num];
9     clique_num ++;
10    delete \( v \) and its neighbors and all edges connecting from/to those vertices from \( G \);
11    find the new vertex \( v \) with the largest degree in the remaining graph \( G \);
12    set \( n = \) this degree number;
13    add all remaining vertices in \( G \) to clique[clique_num];
14 end
```

**Figure 1. Pseudo-code for greedy clique decomposition**

**Definition 2 (Conflict graph).** A conflict graph \( G=(V,E) \) for a CNF formula is defined as follows. Let \( V \) be the set of all its variables\(^4\), for different \( x, y \in V \), edge \( (x, y) \in E \) iff there is a length 2 clause containing exactly these two variables.

From the definition above, it is easy to see that constructing a conflict graph from a given CNF instance only takes time linear to the instance size.

The pseudo code for greedy clique decomposition algorithm is given in Figure 1. Note that this is a straight-forward approximation algorithm. First we find the vertex with the largest degree in the conflict graph (line 4) and then put this vertex and all its neighbors in the conflict graph into the first clique, and then all vertices in the first clique and edges connecting from/to these vertices are removed from the conflict graph, and we find the vertex with the largest degree in the remaining graph (line 5-
10), ..., repeating this procedure until we are left with only non-connected single vertices in the remaining graph, and then all such vertices are treated as a single clique (line 11).

We illustrate the concept of the conflict graph and this approximation algorithm through an example: the pigeon hole problem $PHP^5_4$ with 5 pigeons and 4 holes expressing that we put 5 pigeons into 4 holes without putting two pigeons into one and the same hole, which is obviously unsatisfiable. Let variable $p_{ij}$ denotes that pigeon $i$ is in hole $j$, we encode this problem using the following set of clauses (non-onto version) in Figure 2.

$$
\begin{align*}
&(p_{11} \lor p_{12} \lor p_{13} \lor p_{14} \lor p_{15}) \lor (p_{21} \lor p_{22} \lor p_{23} \lor p_{24} \lor p_{25}) \lor (p_{31} \lor p_{32} \lor p_{33} \lor p_{34} \lor p_{35}) \lor (p_{41} \lor p_{42} \lor p_{43} \lor p_{44} \lor p_{45}), \\
&\neg p_{11} \lor \neg p_{12} \lor \neg p_{13} \lor \neg p_{14} \lor \neg p_{15}, \quad \neg p_{21} \lor \neg p_{22} \lor \neg p_{23} \lor \neg p_{24} \lor \neg p_{25}, \\
&\neg p_{31} \lor \neg p_{32} \lor \neg p_{33} \lor \neg p_{34} \lor \neg p_{35}, \quad \neg p_{41} \lor \neg p_{42} \lor \neg p_{43} \lor \neg p_{44} \lor \neg p_{45}.
\end{align*}
$$

Figure 2. CNF formula of $PHP^5_4$

The corresponding conflict graph is shown in Figure 3. It consists of four cliques:

$$\{p_{11}, p_{12}, ..., p_{1n+1}\}, \{p_{21}, p_{22}, ..., p_{2n+1}\}, \{p_{31}, p_{32}, ..., p_{3n+1}\}, \{p_{41}, p_{42}, ..., p_{4n+1}\}.$$ 

It can be seen that the algorithm correctly decomposes the conflict graph in Figure 3 to these four clique components. One can easily see that for $PHP^m_n$, there will be $n$ cliques in the corresponding conflict graph:

$$\{p_{11}, p_{12}, ..., p_{n+1}\}, \{p_{21}, p_{22}, ..., p_{n+1}\}, ..., \{p_{n1}, p_{n2}, ..., p_{n(n+1)}\},$$

and this algorithm still correctly decomposes the corresponding conflict graph.

For pigeon hole problems, viewing each clique as a bucket, we know theoretically that we will get a polynomial size proof (In this case, the relative order how these buckets are eliminated does not matter.) [8]. Since the worst-case time complexity of this intuitive approximation algorithm is only quadratic to the conflict graph size, we are sure that we can implement the symbolic satisfiability solving algorithm for pigeon hole problems, which is inspiring, since the existing OBDD based solvers [14-16] cannot handle pigeon hole cases efficiently.

It is natural to try this algorithm on other instances of which the conflict graphs are similar to the pigeon hole case, so we can improve the performance of these instances. As our experiments show, this is indeed the case, readers are referred to Section 4 for more details. For the intuitive effectiveness of putting each clique as a bucket when doing variable elimination, we argue that when representing a Boolean function using an OBDD, the more two variables interact with each other in the function, the less size OBDD we can get by putting these variables nearby each other in the variable order. In other words, the dependencies between these variables are the main cause for the OBDD size explosion. It follows that if we can put all these variables into a clique and eliminate them together, it will reduce the sizes of intermediate OBDDs.
3.2 Tree-like computation

For each \( F(X_i) \) mentioned previously, there are two ways of obtaining the corresponding OBDD: linear or tree-like computation. We elaborate this through the following example. Suppose that in order to obtain \( F(X_i) \), we have to join four clauses: \( C_1 \), \( C_2 \), \( C_3 \) and \( C_4 \). The linear and tree-like ways of computation are shown in Figure 4.

The linear computation is straightforward. We start with joining the first two OBDDs, obtaining one temporary result, then join this temporary result with the third OBDD, ..., repeat this procedure until we get the final OBDD. We elaborate the tree-like computation as follows. For a total number of \( 2^n \) leaf nodes, we first obtain \( 2^{n-1} \) temporary results by applying two neighboring nodes, then obtain \( 2^{n-2} \) nodes from these \( 2^{n-1} \) nodes, repeat this procedure until we get the final result node. It must be pointed out that this idea is first used in the source code of Ebddres [15] to the best of our knowledge. It is somehow surprising that in practice we find tree-like computation faster than linear computation. Hence we would like to compare these two methods. In the experimental results, we will see that tree-like computation does give better performance. This may be explained by the fact that in a tree-like computation, the relative size ratio of two OBDDs applying together is more stable than in a linear computation case.

Figure 4. Linear computation (left) and tree-like computation (right)

4. Experimental results

The experiments\(^5\) are carried out on a server machine with 32 GB main memory and four 3 GHz Xeon CPUs running Linux 2.6.9. No parallel mechanism is used. Our implementation makes use of the CUDD package [17]. We use the benchmark suite [18]. Timeout is set to 10 minutes (600 seconds). We chose four state-of-the-art SAT solvers for comparison with our method. They are Ebddres [15], Minisat, Ternary Tree Solver (TTS) [19] and SSAT [9] (We used the latest available versions to the time of writing this paper.).

4.1 Difficult SAT instances selected for experiments

We briefly introduce the problems that we used in our experiment.

**Pigeon Hole Instances (hole).** This is the standard SAT benchmark encoding the pigeon hole principle problem. The problem has been introduced in Section 3. We use 10 instances of this problem ranging from 7 to 30 holes.

**Filed Programmable Gate Array Routing Instances (fpga, chnl).** This benchmark set is obtained by expressing whether it is possible to route \( n \) connection through \( m \) tracks provided by the field programmable gate array component. We use 6 satisfiable and 9 unsatisfiable instances of this problem for various number of required routes and connections. Two different encodings of the problem are used – denoted by fpga and chnl. More details about this benchmark set is provided in [20].

**Randomized Urquhart Instances (Urq).** This set of benchmark problems are hard unsatisfiable instances. More details are provided in [21]. We use 6 instances of the problems of this type.

\(^5\) Our implementation along with the set of benchmarks we used can be downloaded from http://www.utdallas.edu/~wxc103020/bddsat.tar.bz2.
4.2 Results and analysis

Detailed information about the instances we used, the corresponding clique decomposition of their conflict graphs and the experimental results are presented in Table 1. In the first column, we list these instances. We list whether an instance is satisfiable in the second column, and the number of variables and the number of clauses of each instance in the third column. The detailed clique information of corresponding conflict graph of each instance is given in the fourth column (if the instance is clique decomposable). For example, 8*10 indicates that there are 10 cliques of size 8 in the corresponding conflict graph, and 8*10+5*8 indicates that there are 10 cliques of size 8 and 8 cliques of size 5 in the corresponding conflict graph. Careful readers might notice that the number of vertices in this column equals the number of variables of the instance. We would like to point out that for Urq cases, our algorithm fails to find useful clique information and hence as we can see from Table 1, the performance of our implementation is not very good. We present the experimental results from the fifth column to the ninth column. Note that the last column of Table 1 (No.) indicates the correspondence between the instance name and its representation in Figure 5. For example, instance 3 in chnl of Figure 5 is the instance chnl10_13 in Table 1.

<table>
<thead>
<tr>
<th>Instances</th>
<th>Satisfiable</th>
<th>#vars/#clauses</th>
<th>Cliques</th>
<th>Ebddres (s)</th>
<th>Minisat (s)</th>
<th>TTS (s)</th>
<th>SSAT (s)</th>
<th>Our (s)</th>
<th>No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>hole 7</td>
<td>unsat</td>
<td>56/204</td>
<td>8*7</td>
<td>0.00</td>
<td>0.05</td>
<td>0.09</td>
<td>0.01</td>
<td>0.01</td>
<td>7</td>
</tr>
<tr>
<td>hole 8</td>
<td>unsat</td>
<td>72/297</td>
<td>9*8</td>
<td>0.01</td>
<td>0.27</td>
<td>0.28</td>
<td>0.01</td>
<td>0.01</td>
<td>8</td>
</tr>
<tr>
<td>hole 9</td>
<td>unsat</td>
<td>90/415</td>
<td>10*9</td>
<td>0.01</td>
<td>1.56</td>
<td>0.08</td>
<td>0.03</td>
<td>0.02</td>
<td>9</td>
</tr>
<tr>
<td>hole 10</td>
<td>unsat</td>
<td>110/561</td>
<td>11*10</td>
<td>0.14</td>
<td>29.98</td>
<td>0.22</td>
<td>0.05</td>
<td>0.02</td>
<td>10</td>
</tr>
<tr>
<td>hole 11</td>
<td>unsat</td>
<td>132/738</td>
<td>12*11</td>
<td>0.08</td>
<td>598.36</td>
<td>0.43</td>
<td>0.09</td>
<td>0.02</td>
<td>11</td>
</tr>
<tr>
<td>hole 12</td>
<td>unsat</td>
<td>156/949</td>
<td>13*12</td>
<td>0.29</td>
<td>&gt;600.0</td>
<td>1.07</td>
<td>0.14</td>
<td>0.03</td>
<td>12</td>
</tr>
<tr>
<td>hole 13</td>
<td>unsat</td>
<td>182/1197</td>
<td>14*13</td>
<td>1.96</td>
<td>&gt;600.0</td>
<td>2.59</td>
<td>0.25</td>
<td>0.03</td>
<td>13</td>
</tr>
<tr>
<td>hole 14</td>
<td>unsat</td>
<td>210/1485</td>
<td>15*14</td>
<td>1.74</td>
<td>&gt;600.0</td>
<td>7.09</td>
<td>0.44</td>
<td>0.03</td>
<td>14</td>
</tr>
<tr>
<td>hole 20</td>
<td>unsat</td>
<td>420/2221</td>
<td>21*20</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>5.46</td>
<td>0.05</td>
<td>20</td>
</tr>
<tr>
<td>hole 30</td>
<td>unsat</td>
<td>930/33981</td>
<td>31*30</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>54.09</td>
<td>0.20</td>
<td>30</td>
</tr>
<tr>
<td>chnl10_11</td>
<td>unsat</td>
<td>220/1122</td>
<td>11*20</td>
<td>0.24</td>
<td>29.49</td>
<td>0.20</td>
<td>0.23</td>
<td>0.03</td>
<td>1</td>
</tr>
<tr>
<td>chnl10_12</td>
<td>unsat</td>
<td>240/1344</td>
<td>12*20</td>
<td>0.07</td>
<td>49.55</td>
<td>0.23</td>
<td>0.33</td>
<td>0.03</td>
<td>2</td>
</tr>
<tr>
<td>chnl11_13</td>
<td>unsat</td>
<td>260/1586</td>
<td>13*20</td>
<td>0.09</td>
<td>46.90</td>
<td>0.26</td>
<td>0.48</td>
<td>0.03</td>
<td>3</td>
</tr>
<tr>
<td>chnl11_12</td>
<td>unsat</td>
<td>264/1476</td>
<td>12*22</td>
<td>0.13</td>
<td>367.69</td>
<td>0.43</td>
<td>0.46</td>
<td>0.03</td>
<td>4</td>
</tr>
<tr>
<td>chnl11_13</td>
<td>unsat</td>
<td>286/1742</td>
<td>13*22</td>
<td>0.22</td>
<td>&gt;600.0</td>
<td>0.51</td>
<td>0.62</td>
<td>0.03</td>
<td>5</td>
</tr>
<tr>
<td>chnl11_20</td>
<td>unsat</td>
<td>440/2220</td>
<td>20*22</td>
<td>0.81</td>
<td>&gt;600.0</td>
<td>1.02</td>
<td>5.32</td>
<td>0.05</td>
<td>6</td>
</tr>
<tr>
<td>fpga10_8</td>
<td>sat</td>
<td>120/448</td>
<td>8<em>10+5</em>8</td>
<td>0.82</td>
<td>0.00</td>
<td>0.07</td>
<td>N/A</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>fpga10_9</td>
<td>sat</td>
<td>135/549</td>
<td>9<em>10+5</em>9</td>
<td>0.52</td>
<td>4.38</td>
<td>0.15</td>
<td>N/A</td>
<td>0.02</td>
<td>2</td>
</tr>
<tr>
<td>fpga12_8</td>
<td>sat</td>
<td>144/560</td>
<td>8<em>12+6</em>8</td>
<td>2.85</td>
<td>0.00</td>
<td>0.16</td>
<td>N/A</td>
<td>0.03</td>
<td>3</td>
</tr>
<tr>
<td>fpga12_9</td>
<td>sat</td>
<td>162/684</td>
<td>9<em>12+6</em>9</td>
<td>11.05</td>
<td>0.00</td>
<td>0.34</td>
<td>N/A</td>
<td>0.03</td>
<td>4</td>
</tr>
<tr>
<td>fpga12_11</td>
<td>sat</td>
<td>198/968</td>
<td>11<em>12+6</em>11</td>
<td>10.91</td>
<td>&gt;600.0</td>
<td>1.27</td>
<td>N/A</td>
<td>0.03</td>
<td>5</td>
</tr>
<tr>
<td>fpga12_12</td>
<td>sat</td>
<td>216/1128</td>
<td>12<em>12+6</em>12</td>
<td>3.10</td>
<td>49.56</td>
<td>0.61</td>
<td>N/A</td>
<td>0.03</td>
<td>6</td>
</tr>
<tr>
<td>fpga13_9</td>
<td>sat</td>
<td>176/759</td>
<td>9<em>13+7</em>5+6*4</td>
<td>31.55</td>
<td>0.00</td>
<td>0.37</td>
<td>N/A</td>
<td>0.03</td>
<td>7</td>
</tr>
<tr>
<td>fpga13_10</td>
<td>sat</td>
<td>195/905</td>
<td>10<em>13+7</em>5+6*5</td>
<td>37.58</td>
<td>0.00</td>
<td>0.73</td>
<td>N/A</td>
<td>0.03</td>
<td>8</td>
</tr>
<tr>
<td>fpga13_12</td>
<td>sat</td>
<td>234/1242</td>
<td>12<em>13+7</em>6+6*5</td>
<td>46.47</td>
<td>0.00</td>
<td>0.59</td>
<td>N/A</td>
<td>0.03</td>
<td>9</td>
</tr>
<tr>
<td>Urq3_5</td>
<td>unsat</td>
<td>46/470</td>
<td>N/A</td>
<td>0.01</td>
<td>36.40</td>
<td>0.30</td>
<td>&gt;600.0</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>Urq4_5</td>
<td>unsat</td>
<td>74/694</td>
<td>N/A</td>
<td>0.83</td>
<td>&gt;600.0</td>
<td>5.30</td>
<td>&gt;600.0</td>
<td>0.12</td>
<td>2</td>
</tr>
<tr>
<td>Urq5_5</td>
<td>unsat</td>
<td>121/1210</td>
<td>N/A</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>0.93</td>
<td>3</td>
</tr>
<tr>
<td>Urq6_5</td>
<td>unsat</td>
<td>180/1756</td>
<td>N/A</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>0.4</td>
<td>4</td>
</tr>
<tr>
<td>Urq7_5</td>
<td>unsat</td>
<td>240/1294</td>
<td>N/A</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>0.5</td>
<td>5</td>
</tr>
<tr>
<td>Urq8_5</td>
<td>unsat</td>
<td>327/3252</td>
<td>N/A</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>&gt;600.0</td>
<td>0.6</td>
<td>6</td>
</tr>
</tbody>
</table>
Analysis of the results

For each $F(X_i)$ mentioned in Section 3, there are two ways of obtaining the corresponding OBDD: linear computation and tree-like computation. Our implementation (denoted by Our in Table 1) using the greedy clique decomposition can be seen as an advanced version than Ebddres with variable elimination with tree-like computation, in the sense that we use advanced heuristic for dividing all variables into several subgroups (buckets), and we quantify out all variables in each bucket altogether each time. As it is evident from our experimental results, the method brings significant improvement for the selected difficult benchmark problems (hole, chnt, and fpga instances). The improvements are in orders of magnitudes with respect to all tested state-of-the-art SAT solvers. The improvement on selected benchmarks is exponential with respect to the best tested SAT solver.

SSAT and Minisat solvers

SSAT solver is merely a preprocessing tool. It can be seen from the results that putting greedy clique decomposition in the framework of OBDD-based satisfiability solving improves its performance, since we can decide satisfiable instance fast and unsatisfiable instances faster than SSAT. Note that our fpga instances are all satisfiable and therefore SSAT cannot be used to run these instances (see the bottom left plot in Figure 5). Also, the performance of Minisat is in general inferior to other solvers. We believe the reason is that the focus of Minisat is not on small and hard instances, but rather on practical applications. It is also worth noting that the performance of Minisat for fpga instances are rather unstable (see the bottom left plot in Figure 5).

4.3 Experiments of tree-like computation

We also evaluate the performance of tree-like computation and linear computation (We do not use greedy clique decomposition in this experiment.). In this experiment, we run all instances first using tree-like computation and then linear computation. We list the following data: maximum single OBDD size in the case of tree-like computation, maximum single OBDD size and the number of clauses conjioned when the maximum OBDD size is reached in the case of linear computation. We present the results in Table 2.

![Figure 5. Comparison of experimental results](image-url)
We compare tree-like computation and linear computation in Figure 6. We have the following comments regarding the above results, for all instances, although the number of maximum single OBDD size in the tree-like computation case is not very stable (One possible reason for this instability is: since we conjunct two OBDDs in the computation, for those same family of instances but with different sizes, the neighboring clauses which are conjuncted directly in the first instance might not be conjuncted directly in the other instance, and this initial difference might cause the overall maximum single OBDD to be different.), however, except for some very few cases, we can get much smaller max single OBDD size using tree-like computation than linear computation. This also explains why our implementation using the greedy clique plus tree-like computation outperforms most other tools in Table 1.

![Figure 6. Comparison of tree-like computation and linear computation](image-url)
5. Conclusion and future directions

Motivated by recent advances in theoretical results about OBDD proof system, the primary intention of this paper is to show that there is still potential for improvement of the OBDD based satisfiability solving through a rather different top-down approach (from theory to application). We have proposed to explore the cliques in conflict graphs of the instances, and use this information as a variable grouping heuristic when eliminating variables in groups. It must be pointed out that the idea of applying the greedy clique decomposition to satisfiability solving is first proposed by Pavel Surynek in [9], and we apply this idea to symbolic satisfiability solving. Experimental results show that for instances of whose conflict graphs possessing a clique-like structure, this method is quite effective. We hope researches who are interested in SAT and other problems [22, 23] can benefit from these results.

We are aware that restarts facilitated DPLL-based satisfiability solvers are the most competitive solvers in the practical satisfiability field. However, it remains interesting to explore other variants of the satisfiability algorithm rather than only DPLL-based ones since the underlying propositional proof systems behind the practical procedures are different. Moreover, when we intend to explore further on the practical side of a particular propositional proof system, theoretical results can in some sense be a priori guide. As for symbolic satisfiability solving, one next step is to try to make the computation dag-like instead of tree-like since it is already known that the tree-like OBDD proof system with the first three rules cannot polynomially simulate resolution and obviously is weaker than the dag-like OBDD proof system.

6. Acknowledgment

The work is supported by the Beijing Forestry University Young Scientist Fund No. BLX2009013, the Chinese National 973 Plan (No.2010CB328103), the National Natural Science Foundation of China under Grant Nos. 60833001 and 61170028, the CAS Innovation Program and the Fundamental Research Funds for the Central Universities of China (No. JB-GJ1001).

7. References


