Solving Difficult SAT Instances Using Greedy Clique Decomposition

Pavel Surynek

Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic
surynek@ktiml.mff.cuni.cz

Abstract. We are dealing with solving difficult SAT instances in this paper. We propose a method for preprocessing SAT instances (CNF formulas) by using consistency techniques known from constraint programming methodology and by using our own consistency technique based on clique decomposition of a graph representing conflicts in the input formula. The clique decomposition allows us to make a strong reasoning over the SAT instance, which even in some cases decides the SAT instance itself without search. We implemented our preprocessing method in C++ and compared it with several state-of-the-art SAT solvers on the selected difficult SAT instances. The result of application of our method was speedup in orders of magnitudes compared to the tested SAT solvers.

Keywords: SAT, search, consistency, clique, difficult instances

1 Introduction

Our recent work on artificial intelligence planning problems [3] inspired us to exploit our newly developed techniques [27] in solving *Boolean formula satisfaction problems (SAT)*. We were studying the problem of finding supporting actions for a goal in AI planning context. We call this a *supports problem* in short. This is some kind of an important sub-problem which must be solved many times when solving AI planning problems over *planning graphs* [6]. We showed that the supports problem is NP-complete in our work. In doing so we used conversion of an instance of the SAT problem to the instance of the supports problem [27]. This simple proof uncovered some interesting similarities between the SAT problem and the supports problem. Strictly speaking the similarities itself are neither interesting nor useful. They become more interesting after connecting them with our new method for solving supports problems based on a greedy clique decomposition which was also proposed in the mentioned work. The positive experiences made with the method on planning problems and the observed similarity lead us to the idea to adapt the technique of the greedy clique decomposition for solving SAT problems.

Boolean formula satisfaction problems and SAT solving techniques play an extremely important role in theoretical computer science as well as in practice. The question of whether there exist a complete polynomial time SAT solver is a key question for theoretical computer science and is open for many years (the *P vs. NP problem*) [7]. On the other hand the practical use of SAT problems and SAT solvers in real life applications is also very intensive. Applications of SAT solving techniques range from microprocessor verification [29] and field-programmable gate array design [22] to solving AI planning problems by translating them into Boolean formulas [17].

An excellent performance breakthrough was done in solving SAT problems over past years. Thanks to new algorithms and implementation techniques focused on real life SAT problems many of the today's benchmark problems [18, 24] are solved by state-of-the-art solvers [11, 12, 14, 15, 20, 26] in time proportional to the size of the input. It seems that the difficulty of many SAT benchmark problems consists in their size only. Lot of smaller benchmark problems are solved in real-time by today's state-of-the-art SAT solvers. The observation that can be deduced upon these facts is that there is almost no chance to compete with best SAT solvers by own newly written SAT solver on these problems. That is why we are concentrating on difficult instances of SAT problems only, where the word difficult here means difficult for today's state-of-the-art SAT solvers.

A very valuable set of difficult (in the mentioned sense) problems was collected by Aloul [1]. Although these problems are small in length of the input formula they are difficult to be answered. The detailed discussion about hardness of these problems is provided in [2]. However one of the aspects is that these problems are mostly unsatisfiable (and this fact is well hidden in the problem). The solver cannot guess a solution using its advanced techniques and heuristics in such case and must really perform some search in order to prove that there is no solution. In the case of positive answer the satisfying valuation of variables serves a witness (of small size) certifying existence of at least one solution. If the solver obtains a witness its task is complete. In contrast to this, there is no such small witness in the case of negative answer so the search must be performed.

Our contribution to solving SAT problems consists in preprocessing and reformulation of the input Boolean formula in *CNF* form (*conjunctive normal form* - conjunction of disjunctions) which of the result is the answer whether the input formula is unsatisfiable or a new formula with the same set of satisfying valuations as that of the input one. If the input formula is not decided by the preprocessing phase then the preprocessed formula is postponed to the SAT solver of the user's choice. The idea behind this process is to make the task for the SAT solver easier by deciding the input formula even within the fast preprocessing phase or by providing equivalent but simpler formula to the SAT solver. Experiments showed that solving process over mentioned difficult SAT benchmarks speeds up by order of magnitudes by using our approach.

The reformulation within preprocessing phase itself is simple. We are viewing the input Boolean formula in CNF form as a graph with vertexes and edges. For each *literal* (variable or its negation) of the input formula we consider a vertex and for each conflict between literals we consider an edge. Conflicting literals are those that cannot be both satisfied in a single valuation of variables, for example positive and negative literals of the same variable are conflicting. To be able to use our reasoning based on clique decomposition we need a graph with appropriately large *complete subgraphs* (cliques). Unfortunately the graph arising from the above interpretation of the Boolean formula in *CNF* form is rather sparse (the largest clique is of size 2). That is why we apply further inference by which we deduce more conflicts between the literals and which allow us to introduce more edges into the graph. We are using singleton arc-consistency [5] as the inference technique for deducing edges.

Having the graph constructed from the input CNF formula, a clique decomposition of this graph is found by a greedy algorithm (we do not need optimal clique decomposition; we need just some of the reasonable quality). The important property of the constructed clique decomposition is that at most one literal from each clique can be assigned the value $true$. In this situation we perform some kind of literal contribution counting to rule out literals that can never be $true$. To do this, the maximum number of satisfied clauses by literals of each clique is calculated. Then a literal of a certain clique can be ruled out if literals from other cliques together with the selected literal do not satisfy enough clauses to satisfy the input formula.

Although this problem reformulation is looking weak it provides a strong reasoning about dependencies among clauses of the CNF Boolean formula and about the effect of the selection of a value for a variable on the overall satisfiability of the formula. Moreover if all the literals are ruled out during the preprocessing phase the input formula is obviously unsatisfiable. Experimental evaluation showed that this happen on difficult SAT problems very often. Otherwise a new formula in CNF form equivalent to the input one is produced. The new formula is constructed from the original one by adding clauses which capture all the dependencies inferred by the initial singleton arc-consistency stage and by literal contribution counting based on the clique decomposition.

The paper is organized as follows. A detailed formal description of the reformulation of a SAT instance using the greedy clique decomposition is provided in the section 2. The subsequent section 3 is devoted to some experimental comparison of our approach with existing state-of-the-art SAT solvers. We are discussing the contribution of our method within this section too. Finally we put our work in relation to similar works in the field of Boolean satisfiability and propose some future research directions of the studied topic.

2 SAT Reformulation Using Greedy Clique Decomposition

We will formally describe details of the process of SAT problem reformulation in this section. Let $B = \wedge_{i=1}^{n} \vee_{j=1}^{m_i} x_j^i$ be the input Boolean formula in CNF form, where x_j^i is a literal (variable or its negation) for all possible i and j. A sub-formula $\vee_{j=1}^{m_i} x_j^i$ of the input formula B for every possible i is called a clause. The ith clause of the formula B will be denoted as C_i in the following paragraphs. As it was mentioned in the introduction, the basic idea of the SAT problem reformulation consists in viewing the input formula as an undirected graph with vertexes and edges in which the internal structure of the formula is captured in some way. To be concrete the graph will capture pairs of conflicting literals and will be constructed in several stages.

2.1 Inference of Conflicting Literals

We start by the construction of an undirected graph $G_B^1 = (V_B^1, E_B^1)$ which will represent trivially conflicting literals. The graph will be called a *graph of trivial conflicts*. The graph G_B^1 will then undergo further inference process by which additional conflicts will be inferred. We will denote the resulting undirected graph as $G_B^2 = (V_B^2, E_B^2)$ and call it an *intermediate graph of conflicts*.

Construction of the undirected graph G_B^1 is simple. A vertex is introduced into the graph G_B^1 for each literal occurring in the formula B, that is $V_B^1 = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m_i} x_j^i$ (notice that $\left|V_B^1\right|$ is typically smaller than the length of the formula, since literals may occur many times in the formula while only once in the graph). The construction of the set of edges E_B^1 is also straightforward. An edge $\{x_j^i, x_l^k\}$ is introduced into the graph G_B^1 if the literals x_j^i are x_l^k are trivially conflicting, that is if $(x_j^i = v \,\&\, x_l^k = \neg v) \vee (x_j^i = \neg v \,\&\, x_l^k = v)$ for some Boolean variable v. The graph G_B^1 is finished by performing this for all possible pairs of conflicting literals. The interpretation of the graph of conflicts is that if a literal corresponding to a vertex is selected to be assigned the value *true* all literals corresponding to the neighboring vertexes must be assigned the value *false*.

A real-life graph resulting from the described process over a selected benchmark problem is shown in the left part of figure 1. The resulting graph is evidently sparse, since there are edges only between literals of the same variable. As it is not a good starting point for our method further inference mechanism for discovering more conflicting pairs of literals (more edges for the graph) must be applied. This further inference mechanism takes the already constructed graph G_B^1 and augments it by adding new edges, the result of this stage is the intermediate graph of conflicts G_B^2.

The process of construction of the graph G_B^2 exploits techniques known from standard SAT resolution approaches and from *constraint programming* [9] - *unit propagation* [10, 30], *arc-consistency* (*AC*) [19] and *singleton arc-consistency* (*SAC*) [5]. Before describing the construction of the graph G_B^2 let us recall these notions. While doing this we will adapt notations of these concepts slightly for the SAT domain to prepare them for our purposes. The following definitions assume the input formula B in CNF form and a corresponding graph of conflicts G_B (for example the graph G_B^1 expressing trivial conflicts).

Definition 1 (Arc-consistency in SAT instance with respect to the graph of conflicts). Consider two clauses C_i and C_k for $i,k \in \{1,2,\ldots,n\}$ of the formula B. A literal x_j^i ($j \in \{1,2,\ldots,m_i\}$) from the clause C_i is *supported* by the clause C_k with respect to the given graph of conflicts G_B if there exists a literal x_l^k ($l \in \{1,2,\ldots,m_k\}$) from the clause C_k, such that the literals x_j^i and x_l^k are not in a conflict with respect to the graph G_B. An ordered pair of clauses (C_i,C_k) of the formula B is called an *arc* in this context. An arc (C_i,C_k) for some $i,k \in \{1,2,\ldots,n\}$ is *consistent* (*or arc-consistent*) with respect to the graph of conflicts G_B if all the literals of the clause C_i are supported by the clause C_k with respect to the graph of conflicts G_B. The formula B is called *arc-consistent* with respect to the graph conflicts G_B if all the arcs (C_i,C_k) for all $i,k = 1,2,\ldots,n$ are arc-consistent with respect to the graph of conflicts G_B. o

The reason for having the definition of arc-consistency is that the literals which are not supported according to the definition cannot be assigned the value *true* (this means that the corresponding variable cannot be assigned the value *false* in the case of negative literal). So the solver can rule out such literals from further attempts to assign them the value *true*, which may reduce the size of the search space. Notice that the definition has the graph of conflicts G_B as a parameter. It is possible to put any correct graph of conflicts as a parameter of this definition, whereas correct means, that if $\{y,z\}$ is the edge of the graph then $B \Rightarrow y \neq z$ must be a tautology. This is obviously true for the graph of trivial conflicts G_B^1. Notice also that if we use the graph of trivial conflicts G_B^1 the definition becomes identical to unit propagation [10, 30].

Having the Boolean formula B the question is how to make it arc-consistent with respect to the given graph of conflicts. For this purpose we adopt techniques developed in constraint programming and by SAT community, namely the arc-consistency enforcing algorithms [9, 19] and unit propagation [10, 30]. There is a great variety of such algorithms, however their common feature is the search for supports for every value (literal) which is suspected of not being supported. The main difference among these algorithms is the efficiency of the search for supports. If an unsupported literal is detected it is ruled out. Ruling out an unsupported literal may cause that some other literal lose its support. This chain like propagation of changes continues until a stable state is reached. For purposes of SAT domain this propagation process is usually augmented by an additional simplification rule. If the consistency enforcing algorithm detects that within some clause there is only one literal that can be selected to be *true*, it is fixed to the value *true* and the corresponding clause is cut out from further reasoning (it is exactly the simplification rule from unit propagation).

Unfortunately the defined arc-consistency over Boolean formulas in the CNF form is too weak to infer significantly more conflicts than they are already present in the graph of trivial conflicts. Therefore we need to make the consistency stronger. Perhaps the simplest way to do this is to make the selected consistency technique *singleton* [5]. The following definition again assumes the Boolean formula B and the corresponding graph of conflicts G_B (again graph of trivial conflicts G_B^1 can be used).

Definition 2 (Singleton arc-consistency in SAT instance with respect to the graph of conflicts). A literal x_l^k ($l \in \{1,2,\ldots,m_k\}$) from a clause C_k for $k \in \{1,2,\ldots,n\}$ of the formula B is *singleton arc-consistent* with respect to the given graph of conflicts G_B if the formula obtained from B by replacing the clause C_k by the literal x_l^k (the resulting formula is $(\wedge_{i=1}^{k-1} \vee_{j=1}^{m_i} x_j^i) \wedge x_l^k \wedge \wedge(\wedge_{i=k+1}^{n} \vee_{j=1}^{m_i} x_j^i)$) is arc-consistent with respect to the graph of conflicts G_B. o

Unsupported literals in the formula modified by replacing the clause C_k by the literal x_l^k are in conflict with the literal x_l^k. This is quite intuitive, the selection of the literal x_l^k to be assigned the value *true* rules out some other literals. Hence these literals are in conflict with the selected literal x_l^k. Having singleton arc-consistency we are ready to infer new edges for graph of conflicts.

The intermediate graph of conflicts G_B^2 is constructed from the graph of trivial conflicts G_B^1 in the following way. Initially the graph G_B^2 is same as the graph G_B^1, that is we start with the initialization $V_B^2 \leftarrow V_B^1$ and $E_B^2 \leftarrow E_B^1$. Then for every literal $y \in V_B^2$ singleton arc-consistency with respect to the graph of conflicts G_B^1 is enforced. If the consistency discovers some unsupported literals, say literals $z_1, z_2, \ldots, z_m$, edges $\{y, z_i\}$ for all $i = 1,2,\ldots,m$ are added into the set of edges E_B^2.

An example of the resulting graph of conflicts is shown in the right part of the figure 1. It is constructed from the original graph of trivial conflicts from the left part of the figure 1. The required complete sub-graphs of the graph are clearly visible.

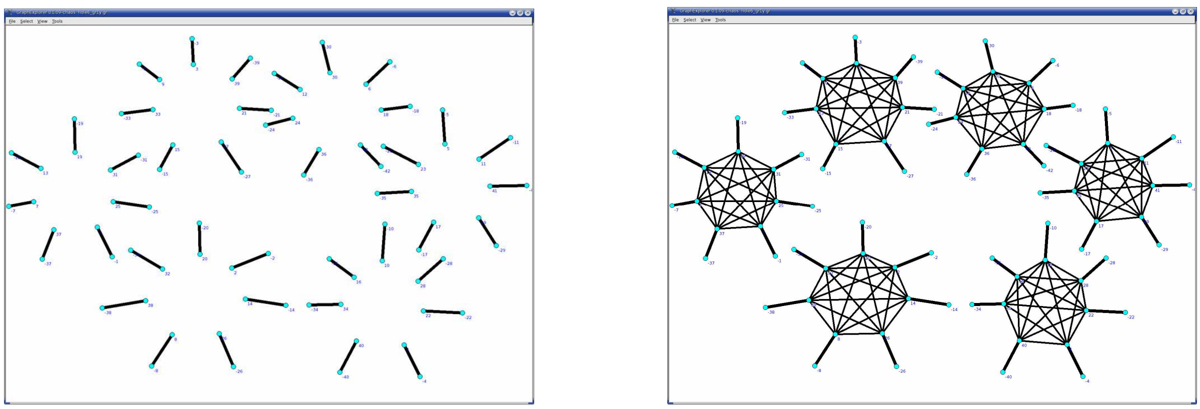

Fig. 1. Left part of the figure shows a graph of trivial conflicts for the SAT benchmark problem pigeon-hole principle number 6 (hole06.cnf). Vertexes represents literals, edges are between pairs of positive and negative literals of the same variable. Right part of the figure shows an intermediate graph of conflicts inferred from the original graph from the left by singleton arc-consistency. The graph contains edges from the original graph plus the inferred edges. Six complete sub-graphs each containing seven vertexes are clearly visible and can be found by a simple greedy algorithm.

The described process of inference of conflicting literals is relatively generic. Both alternative initial graphs of trivial conflicts as well as alternative consistency techniques to arc-consistency and singleton arc-consistency for inference of new edges can be used. Both entities may be considered as parameters of the method.

2.2 Greedy Clique Decomposition and Literal Contribution Counting

To deduce yet more information from the graph of conflicts $G_B^2 = (V_B^2, E_B^2)$ a clique decomposition of the graph is constructed. Formally said a partition of vertexes $V_B^2 = K_1 \cup K_2 \cup \ldots \cup K_s$ such that each set of vertexes K_i for $i = 1,2,\ldots,s$ induces a clique over the set of edges E_B^2 and $K_i \cap K_j = \emptyset$ for all $i, j = 1,2,\ldots,s$ & $i \neq j$. Let E_{K_i} denotes the set of edges induced by the clique K_i, let E_R denotes the set of edges outside the clique decomposition, that is $E_R = E_B^2 - \bigcup_{i=1}^{s} E_{K_i}$. Our inference method based on literal contribution counting requires cliques of the decomposition to be largest as possible (that is s must be smallest as possible) and the size of E_R to be smallest as possible. The better the quality of the decomposition is the stronger results are produced by our inference method. Since the problem of finding optimal clique decomposition with respect to the above criterion is obviously *NP*-complete on a general graph [16], we cannot afford to construct optimal decomposition and must abandon this requirement. Nevertheless experiments showed that the simple greedy algorithm can find a clique decomposition of acceptable quality (with respect to clique sizes and the number of edges outside the decomposition).

Our greedy algorithm for finding clique decomposition is based on the standard greedy algorithm for finding the largest clique. A vertex of the highest degree is found in the graph and is included into the constructed clique. Then the graph is restricted on the neighborhood of the selected vertex and a vertex of the highest degree in this neighborhood is selected as second. Then the graph is again restricted on the neighborhood of these two vertexes (that is considered vertexes are neighbor of both the first and the second selected vertex) and algorithm continues until the neighborhood of selected vertexes is non-empty. The constructed clique and its neighborhood is removed from the graph and the next clique is constructed. This main loop continues until the graph is non-empty.

The described greedy algorithm performed over the graph from the right part of the figure 1 finds the clique decomposition consisting of six cliques of size seven. The fact that at most one literal from a clique can be selected to be assigned the value *true* is used in our inference method.

For the following definitions us have a Boolean formula $B = \wedge_{i=1}^{n} \vee_{j=1}^{m_i} x_j^i$ and the corresponding clique decomposition $V_B^2 = K_1 \cup K_2 \cup \ldots \cup K_s$ of the intermediate graph of conflicts $G_B^2 = (V_B^2, E_B^2)$. Next let $I \subseteq \{1,2,\ldots,n\}$ be a set of indexes of clauses of the formula B. The set I defines a sub-formula B_I of the formula B, where $B_I = \wedge_{i \in I} C_i$.

Definition 3 (Literal contribution). A *contribution of a literal* y to the sub-formula B_I is defined as $|\{i \in I \mid y \in C_i\}|$ and is denoted as $c(y, I)$. ο

Definition 4 (Clique contribution). A *contribution of a clique* $K \in \{K_1, K_2, \ldots, K_s\}$ to the sub-formula B_I is defined as $\max_{y \in K}(c(y, I))$ and is denoted $c(K, I)$. ο

The concept of clique contribution is helpful when we are trying to decide whether it is possible to satisfy the sub-formula B_I using the literals from the clique decomposition. If for instance $\sum_{i \in I} c(K_i, I) < |I|$ holds then the sub-formula B_I cannot be satisfied and hence also B cannot be satisfied. Moreover we can handle a more general case as it is described in the following definitions.

Definition 5 (Clique-consistent literal). A literal $y \in K_i$ for $i \in \{1,2,\ldots,n\}$ is said to be *clique-consistent with respect to the sub-formula* B_I if $\sum_{j \in I \ \& \ j \neq i} c(K_j, I) \geq |I| - c(y, I)$. o

Definition 6 (Clique-consistent formula). A formula B is *clique-consistent with respect to the sub-formula* B_I if all the literals of the formula B are clique-consistent with respect to B_I. o

It is easy to see that a clique-inconsistent literal with respect to some sub-formula of B cannot be selected to be assigned the value $true$. Thus such literals can be ruled out from further reasoning. The proof of this claim is provided in technical report [27]. In addition this type of consistency is strictly stronger than the discussed unit propagation, arc-consistency and singleton arc-consistency. The proof of this claim is again provided in [27].

The remaining question is now how to select the described sub-formulas B_I of B which are used for computation of clique-inconsistent literals. This selection is crucial for the strength of the proposed clique-consistency. It is expectable that we need to rule out as many as possible inconsistent literals. As it is impossible to compute the defined consistency with respect to all such sub-formulas of B, because they are too many, we need to select them with care. The experimentation carried out in [27] shows that a good strength of the clique-consistency can be obtained by selecting clauses into the sub-formula B_I which have the same number of literals. More precisely, we use sub-formulas $B_{I_r} = \wedge_{i \in I_r} C_i$ of B, where $I_r = \{i \in \{1,2,\ldots,n\} \mid m_i = r\}$ for all possible $r \in \mathbb{N}$ for which B_{I_r} is not empty (we suppose that a clause of B does not contain an individual literal more than once). Let us note that we do not know whether this selection is the best possible.

Theorem 1 (Complexity of clique-consistency enforcing algorithm). *There exists a polynomial time algorithm for enforcing clique-consistency with respect to a sub-formula of a given input formula.* n

The proof of this theorem can be found in [27]. Having such algorithm it is possible to extend it for multiple sub-formulas B_{I_r} simply by running the algorithm for each $r \in \mathbb{N}$ for which B_{I_r} is non-empty. Since r is proportional to the size of the input the resulting algorithm is also polynomial.

2.3 Output of the Reformulation Process

At this point everything is ready to introduce the final step of our reformulation method. We will be constructing a modified formula β which is initially set to be same as B. We will further preprocess B by the singleton version of the defined clique-consistency. Conflicts inferred by this further preprocessing will be stored into a new graph of conflicts $G_B^3 = (V_B^3, E_B^3)$ which is initially set to be same as the graph G_B^2. The graph G_B^3 will be called a *final graph of conflicts* in this context.

Singleton clique-consistency is computed in the following way. For each literal y of the input formula B we enforce clique-consistency for the formula obtained from B by selecting the literal y to be assigned the value $true$. More precisely, clauses containing y are removed and negation of the literal y is removed from remaining clauses of B (removal of a literal x_k^i from the clause

$C_i = \vee_{j=1}^{m_i} x_j^i$ of the formula B is defined as replacement of the clause C_i by the clause $(\vee_{j=1}^{k-1} x_j^i) \vee (\vee_{j=k+1}^{m_i} x_j^i))$. The clique-consistency is then enforced for the resulting formula. During enforcing consistency some literals may be found inconsistent. These literals are in conflict with the literal y. If for some clause all its literals are found inconsistent with y then the literal y cannot be selected to be *true* and a new clause $\neg y$ is added to β ($\beta \leftarrow \beta \wedge \neg y$). Otherwise conflicting literals are stored into the graph of conflicts G_B^3 as new edges (that is if the literal y is in conflicts with the literal z, the edge $\{y,z\}$ is included into G_B^3).

If for some clause it is discovered by the clique-consistency that no of its literals can be assigned the value *true* the process terminates with the answer that the formula B cannot be satisfied. This outcome is ensured by the correctness of the method. Our experiments showed that this situation is the most successful case, because an answer to the satisfiability is obtained in polynomial time without any expensive search for solution.

If the process does not terminate with the negative answer all the edges of the graph of conflicts G_B^3 are translated into new clauses of the formula β. That is for every edge $\{y,z\} \in E_B^3$ we add a clause $y \vee \neg z$ into the formula β ($\beta \leftarrow \beta \wedge (y \vee \neg z)$). The resulting formula β is equivalent with the original input formula B. Notice that conflicts inferred by the preceding reformulation stages are also reflected in the formula β, since the graph G_B^3 subsumes the preceding graphs of conflicts G_B^1 and G_B^2. The formula β is finally postponed to the SAT solver of the user's choice. Justification of this step is provided by the following corollary of the correctness of the clique-consistency.

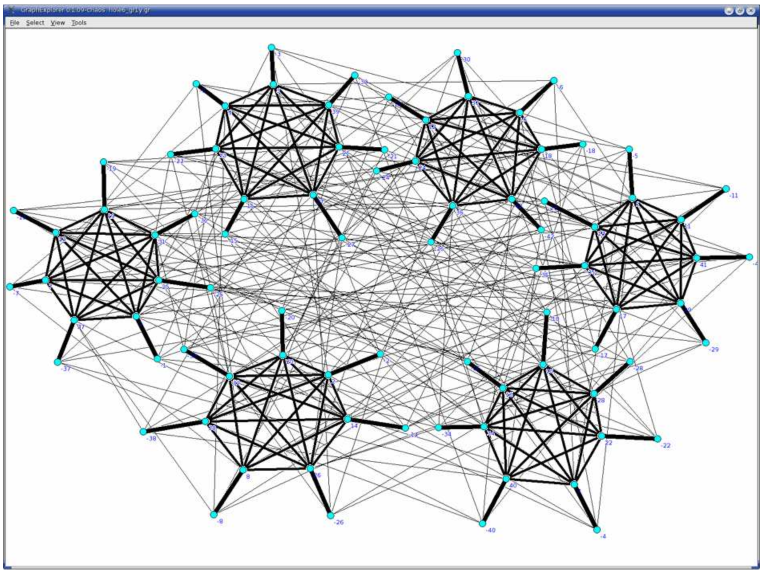

Fig. 2. A final graph of conflicts for the SAT benchmark problem pigeon-hole principle number 6 (hole06.cnf). The graph contains edges from the intermediate graph of conflicts from figure 1 plus the edges inferred by singleton clique-consistency.

Corollary 1 (Correctness of reformulation). *The formula β resulting from the described pre-processing has the same set of satisfying valuations as the original formula B.* ■

The graph of conflicts G_B^3 resulting from processing the intermediate graph of conflicts G_B^2 for the SAT benchmark problem from figure 1 is shown in figure 2.

3 Experimental Results

We chose three state-of-the-art SAT solvers for comparison with our reformulation method. The SAT solvers of our choice were zChaff [14, 20], HaifaSAT [15, 26] and MiniSAT [11, 12] (we used the latest available versions to the time of writing this paper). Our choice was guided by the results of several last SAT competitions [18, 24] in which these solvers numbered among winners. The secondary guidance was that complete source code (in C/C++) for all these solvers is available on web pages of their authors. As we implemented our method in C++ too, this fact allowed us to compile all source codes by the same compiler with the same optimization options which guarantees more equitable conditions for comparison (complete source code implementing our method in C++ can be found at the web page: http://ktiml.mff.cuni.cz/~surynek/software/ssat/ssat.html). All the tests were run on a machine with two AMD Opteron 242 processors (1600 MHz) with 1GB of memory under Mandriva Linux 10.2. Our method as well as the listed SAT solvers were compiled by the gcc compiler version 3.4.3 with options provided maximum optimization for the target testing machine (-O3 -mtune=opteron). Although the testing machine has two processors no parallel processing was used.

3.1 Difficult SAT Instances Selected for Experiments

The testing set consisted of several difficult unsatisfiable SAT instances. This set of benchmark problems was collected by Aloul [1] and is provided at his research web page. The details about hardness and construction of these instances are discussed in [2]. Though let us briefly introduce the problems.

Pigeon Hole Instances. *[hole]* This is the standard SAT benchmark encoding the pigeon hole principle problem. The problem asks whether it is possible to place $n+1$ pigeons in n holes without two pigeons being in the same hole. The problem is obviously unsatisfiable. We used six instances of this problem ranging from 6 to 12 holes.

Randomized Urquhart Instances. *[urq]* This set of benchmark problems contains several artificially constructed hard unsatisfiable instances. More details about these problems are provided in [28]. In addition the problems were randomized for our testing purposes. We used four instances of the problems of this type.

Field Programmable Gate Array Routing Instances. *[fpga, chnl]* This benchmark problem resembles the pigeon hole problem. The question is whether it is possible to route n connection through m tracks provided by the field programmable gate array component. If $n > m$ the problem cannot be satisfied. We used sixteen unsatisfiable instances of this problem for various number of required routes and connections. Two different encodings of the problem are used - denoted *fpga* and *chnl*. More details about encoding of this problem is provided in [22].

Table 1. Experimental comparison of three SAT solvers over selected difficult benchmark SAT instances. We used the timeout of 10.0 minutes (600.00 seconds) for all the tests.

Instance	**Satisfiable**	**Number of variables / number of clauses**	**MiniSAT (seconds)**	**zChaff (seconds)**	**HaifaSAT (second)**
chnl10_11	unsat	220/1122	**34.30**	**7.54**	*> 600.00*
chnl10_12	unsat	240/1344	**101.81**	**9.11**	*> 600.00*
chnl10_13	unsat	260/1586	**200.30**	**11.47**	*> 600.00*
chnl11_12	unsat	264/1476	*> 600.00*	**33.49**	*> 600.00*
chnl11_13	unsat	286/1472	*> 600.00*	**187.08**	*> 600.00*
chnl11_20	unsat	440/4220	*> 600.00*	**329.57**	*> 600.00*
urq3_5	unsat	46/470	**95.04**	*> 600.00*	*> 600.00*
urq4_5	unsat	74/694	*> 600.00*	*> 600.00*	*> 600.00*
urq5_5	unsat	121/1210	*> 600.00*	*> 600.00*	*> 600.00*
urq6_5	unsat	180/1756	*> 600.00*	*> 600.00*	*> 600.00*
hole6	unsat	42/133	**0.01**	**0.01**	**0.01**
hole7	unsat	56/204	**0.09**	**0.04**	**0.02**
hole8	unsat	72/297	**0.49**	**0.23**	**0.94**
hole9	unsat	90/415	**3.64**	**1.46**	**478.16**
hole10	unsat	110/561	**39.24**	**7.53**	*> 600.00*
hole11	unsat	132/738	*> 600.00*	**32.36**	*> 600.00*
hole12	unsat	156/949	*> 600.00*	**372.18**	*> 600.00*
fpga10_11	unsat	220/1122	**44.77**	**12.58**	*> 600.00*
fpga10_12	unsat	240/1344	**119.26**	**33.82**	*> 600.00*
fpga10_13	unsat	260/1586	**362.24**	**76.15**	*> 600.00*
fpga10_15	unsat	300/2130	*> 600.00*	**274.84**	*> 600.00*
fpga10_20	unsat	400/3840	*> 600.00*	**546.00**	*> 600.00*
fpga11_12	unsat	264/1476	*> 600.00*	**55.70**	*> 600.00*
fpga11_13	unsat	286/1742	*> 600.00*	**237.54**	*> 600.00*
fpga11_14	unsat	308/2030	*> 600.00*	*> 600.00*	*> 600.00*
fpga11_15	unsat	330/2340	*> 600.00*	*> 600.00*	*> 600.00*
fpga11_20	unsat	440/4220	*> 600.00*	*> 600.00*	*> 600.00*

For each benchmark SAT instance we measured the overall time necessary to decide its satisfiability. The results are shown in table 1 and table 2. The speedup obtained by using our method compared to a selected SAT solver is also shown.

Table 2. Experimental comparison of three SAT solvers with the method using clique-consistency over selected difficult benchmark SAT instances. Again timeout of 10.0 minutes (600.00 seconds) for all the tests was used.

Instance	Decided by preprocess-ing	Cliques (count x size)	Decision (seconds)	Speedup ratio w.r.t. MiniSAT	Speedup ratio w.r.t zChaff	Speedup ratio w.r.t HaifaSAT
chnl10_11	yes	20 x 11	**0.43**	**79.76**	**17.53**	*> 1395.34*
chnl10_12	yes	20 x 12	**0.60**	**169.68**	**8.51**	*> 1000.00*
chnl10_13	yes	20 x 13	**0.78**	**256.79**	**14.70**	*> 769.23*
chnl11_12	yes	22 x 12	**0.70**	*> 857.14*	**47.84**	*> 857.14*
chnl11_13	yes	22 x 13	**0.92**	*> 652.17*	**203.34**	*> 652.17*
chnl11_20	yes	22 x 20	**5.74**	*> 104.42*	**57.41**	*> 104.42*
urq3_5	no	47 x 2	**130.15**	**0.73**	*N/A*	*N/A*
urq4_5	no	73 x 2	*> 600.00*	*N/A*	*N/A*	*N/A*
urq5_5	no	120 x 2	*> 600.00*	*N/A*	*N/A*	*N/A*
urq6_5	no	179 x 2	*> 600.00*	*N/A*	*N/A*	*N/A*
hole6	yes	6 x 7	**0.01**	**1.0**	**1.0**	**1.0**
hole7	yes	7 x 8	**0.02**	**4.5**	**2.0**	**1.0**
hole8	yes	8 x 9	**0.04**	**12.25**	**5.75**	**23.5**
hole9	yes	9 x 10	**0.08**	**45.5**	**18.25**	**5977.00**
hole10	yes	10 x 11	**0.13**	**301.84**	**57.92**	*> 4615.38*
hole11	yes	11 x 12	**0.20**	*> 3000.00*	**161.8**	*> 3000.00*
hole12	yes	12 x 13	**0.30**	*> 2000.00*	**1240.6**	*> 2000.00*
fpga10_11	yes	20 x 11	**0.46**	**97.32**	**27.34**	*> 1304.34*
fpga10_12	yes	20 x 12	**0.64**	**186.34**	**52.84**	*> 937.50*
fpga10_13	yes	20 x 13	**0.84**	**431.23**	**90.65**	*> 714.28*
fpga10_15	yes	20 x 15	**1.39**	*> 431.65*	**197.72**	*> 431.65*
fpga10_20	yes	20 x 20	**4.72**	*> 127.11*	**115.67**	*> 127.11*
fpga11_12	yes	22 x 12	**0.76**	*> 789.47*	**73.28**	*> 789.47*
fpga11_13	yes	22 x 13	**1.01**	*> 594.05*	**235.18**	*> 594.05*
fpga11_14	yes	22 x 14	**1.30**	*> 461.53*	*> 461.53*	*> 461.53*
fpga11_15	yes	22 x 15	**1.67**	*> 359.28*	*> 359.28*	*> 359.28*
fpga11_20	yes	22 x 20	**5.96**	*> 100.67*	*> 100.67*	*> 100.67*

3.2 Effect of Problem Reformulation

As it is evident from our experimentation the proposed method brings significant improvement in term of time necessary for decision of the selected difficult benchmark problems (Pigeon hole, FPGA routing instances). The improvements are in orders of magnitudes with respect to all tested state-of-the-art SAT solvers. It seems that the improvement on selected benchmarks is exponential with respect to the best tested SAT solver. The conclusion is that there is still room to improve SAT solvers. However the domain of the improvement is more likely over difficult instances of SAT problems which are typically unsatisfiable. It also evident that the clique-consistency is not an universal method for difficult SAT instances. There is no improvement on instances where no cliques of reasonable size are found (randomized Urquhart instances). The interesting feature of the tested SAT instances is that they contain cliques of the same size. This may be accounted to the symmetrical formulation of the problems.

In our minor experiments we also performed the presented experiments with RSAT solver [23]. The results were very similar in the sense that the solver does not cope well with these problems. Unfortunately the solver is provided without the source code as an executable only so we do not consider this test as relevant one. Another SAT solver which worth consideration for our tests (achieved good results in the SAT Race competition [24]) - Eureka [21] - is not provided at all (no source code nor executables are provided).

We also tested our approach on SAT instances where the preprocessing stage does not terminate by the answer that the given SAT instance cannot be satisfied. This is the situation when the problem is not decided by the preprocessing stage and new equivalent SAT instance is produced and postponed to the solver. In such situations our method does not provide competitive results. The resulting formula is typically solved slightly faster but the preprocessing stage takes too much time. The unaffordable time consummation in the preprocessing stage is caused by extensive propagation performed by the method by which huge numbers of conflicts are inferred. It seems that on these problems the proposed approach is too strong and represents an overhead only. The numbers of inferred conflicts is not proportional to the time saved in the search for solution stage. But this is expectable. Moreover, as it was mentioned in the introduction, there is almost no room for improving SAT solvers on such easy (satisfiable) SAT instances.

The question may be now what to do when we have a new problem of unknown difficulty. That is shall we use our preprocessing method or the SAT solver of our choice directly ? The answer is easy. We can run both the preprocessing method and the SAT solver in parallel. On a machine with more than one processor we obtain an exponential speedup (the method succeeds) or no improvement. On a machine with only one processor we may obtain an exponential speedup at the expense of constant slowdown (where the constant is approximately 2).

3.3 Implementation Issues

Although we obtained significant speedup compared to the tested SAT solvers on selected SAT instances we presume that the speedup can be yet improved by a better implementation of our preprocessing method. Our current implementation is an experimental prototype and the quality of our code is uncompetitive with the quality of code of the tested SAT solvers.

4 Related Works

Our method for SAT problem reformulation was originally proposed for solving planning problems over planning graphs. It was named projection consistency and it was described in the technical report [27] by Surynek. The clique-consistency proposed in this paper is an adaptation of projection consistency for the SAT domain.

The idea of exploiting structural information for solving problems is not new. There is lot of works concerning this topic. Many of these works are dealing with methods for breaking symmetries [2, 4, 8]. We share the goal with these methods, which is to reduce the search space. However we differ in the way how we are doing this. We are rather trying to infer what would happen if the search over the problem proceeds in some way. And if that direction seems to be unpromising the corresponding part of the search space is skipped. Symmetry breaking methods are rather trying no to do the same work twice (or more times) by clever a transformation of the original problem.

Our work was much influenced by the paper of Aloul, Markov and Sakallah [2]. We are studying the same set of difficult SAT problems. However it seems that our method is simpler to implement and more effective on the set of selected testing problems.

Finally let us note that the detection of cliques in the structure of the problem is not new. A work dealing with a consistency based on cliques of inequalities was published by Sqalli and Freuder [25]. They use information about cliques to reach more global reasoning about the problem. Another work dealing with the similar ideas is [13] in which the authors use graph structure of the problem to transform it into another formulation based on global constraints, which provide stronger propagation that the original formulation.

5 Conclusions and Future Work

We proposed a method for preprocessing difficult (unsatisfiable) SAT instances based on the greedy clique decomposition of the transformed input CNF formula. Although the method is not universal it provides improvements in orders of magnitudes compared to the state-of-the-art SAT solvers on tested SAT instances. Moreover our method can be easily integrated into a SAT solver (new or existing) which may significantly improve its performance on difficult SAT instances.

For future we plan to further tune the method to be able to cope better with the problems having few edges in the graphs of conflicts (for example Urquhart instances). This may be done by some alternative consistency technique instead of singleton arc-consistency. We also plan to investigate the possibility to make the preprocessing iterative. That is to further preprocess the formula resulting from the previous preprocessing.

We also plan to write an experimental SAT solver which would utilize the clique-consistency during search. This may be useful for early determining that a certain part of the search space does not contain a solution.

Finally the interesting research direction is some kind of a combination of existing symmetry breaking methods and the proposed clique-consistency.

References

1. Aloul, F. A.: Fadi Aloul's Home Page - SAT Benchmarks. Personal Web Page. http://www.eecs.umich.edu/~faloul/benchmarks.html, University of Michigan, USA, (March 2007).

2. Aloul, F. A., Ramani, A., Markov, I. L., Sakallah, K. A.: Solving Difficult SAT Instances in the Presence of Symmetry. Proceedings of the 39th Design Automation Conference (DAC-2002), 731-736, USA, ACM Press, 2002.

3. Allen, J., Hendler, J., Tate, A. (editors): Readings in Planning. Morgan Kaufmann Publishers, 1990.

4. Benhamou, B., Sais, L.: Tractability through Symmetries in Propositional Calculus. Journal of Automated Reasoning, volume 12-1, 89-102, Springer-Verlag, 1994.

5. Bessière, C., Debruyne, R.: Optimal and Suboptimal Singleton Arc Consistency Algorithms. Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-2005), 54-59, Canada, Professional Book Center, 2005.

6. Blum, A. L., Furst, M. L.: Fast Planning through Planning Graph Analysis. Artificial Intelligence 90, 281-300, AAAI Press, 1997.

7. Cook, S. A.: The Complexity of Theorem Proving Procedures. Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 151-158, USA, ACM Press, 1971.

8. Crawford, J. M., Ginsberg, M. L., Luks, E. M., Roy, A.: Symmetry-Breaking Predicates for Search Problems. Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR-96), 148-159, Morgan Kaufmann, 1996.

9. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers, 2003.

10. Dowling, W., Gallier, J.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. Journal of Logic Programming, 1(3), 267-284, Elsevier, 1984.

11. Eén, N., Sörensson, N.: MiniSat — A SAT Solver with Conflict-Clause Minimization. Poster, 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005), Scotland, 2005.

12. Eén, N., Sörensson, N.: The MiniSat Page. Research Web Page. http://www.cs.chalmers.se/Cs/Research/ FormalMethods/MiniSat/Main.html, Chalmers University, Sweden, (March 2007).

13. Frisch, A. M., Miguel, I., Walsh, T.: CGRASS: A System for Transforming Constraint Satisfaction Problems. Barry O'Sullivan (Editor): Recent Advances in Constraints, 15-30, LNCS 2627, Springer-Verlag, 2003.

14. Fu, Z., Marhajan, Y., Malik, S.: zChaff. Research Web Page. http://www.princeton.edu/ ~chaff/ zchaff.html, Princeton University, USA, (March 2007).

15. Gershman, R., Strichman, O.: HaifaSat – a new robust SAT solver. Research Web Page. http://www.cs.technion.ac.il/~gershman/HaifaSat.htm, Technion Haifa, Israel, (March 2007).

16. Golumbic, M. C.: Algorithmic Graph Theory and Perfect Graphs. Academic Press, 1980.

17. Kautz, H. A., Selman, B.: Planning as Satisfiability. Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92), 359-363, Austria, John Wiley and Sons, 1992.

18. Le Berre, D., Simon, L.: SAT Competition 2005. Competition Web Page, http://www.satcompetition.org/2005/, Scotland, (March 2007).

19. Mackworth, A. K.: Consistency in Networks of Relations. Artificial Intelligence 8, 99-118, AAAI Press, 1977.

20. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. Proceedings of the 38th Design Automation Conference (DAC-2001), 530-535, USA, ACM Press, 2001.

21. Nadel, A.: Alexander Nadel's Page. Research Web Page. http://www.cs.tau.ac.il/~ale1/, Tel-Aviv University, Israel, (March 2007).

22. Nam, G.-J., Sakallah, K. A., Rutenbar, R.: A New FPGA Detailed Routing Approach via Search-Based Boolean Satisfiability. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, volume 21-6, 674-684, IEEE Press, 2002.

23. Pipatsrisawat, K., Darwiche, A.: RSat - ...veRSATile... Research Web Page. http://reasoning.cs.ucla.edu/rsat/, University of California Los Angeles, USA, (March 2007).

24. Sinz, C.: SAT-Race 2006. Competition Web Page, http://fmv.jku.at/sat-race-2006/, USA, (March 2007).

25. Sqalli, M. H., Freuder, E. C.: Inference-Based Constraint Satisfaction Supports Explanation. Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference (AAAI-96 / IAAI-96), 318-325, AAAI Press / The MIT Press, 1996.

26. Strichman, O., Gershman, R.: HaifaSat: a New Robust SAT Solver. Proceedings of the 1st International Haifa Verification Conference, LNCS 3875, 76-89, Israel, Springer-Verlag. 2005.

27. Surynek, P.: Projection Global Consistency: An Application in AI Planning. Technical report, ITI Series, 2007-333, http://iti.mff.cuni.cz/series, Charles University, Prague, Czech Republic, 2007.

28. Urquhart, A.: Hard Examples for Resolution. Journal of the ACM, volume 34, 209-219, ACM Press, 1987.

29. Velev, M. N., Bryant, R. E.: Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. Journal of Symbolic Computation (JSC), volume 35-2, 73-106, Elsevier, 2003.

30. Zhang, H., Stickel, M.: An efficient algorithm for unit-propagation. Proceedings of the 4th International Symposium on Artificial Intelligence and Mathematics, USA, 1996.