BDD-Guided Clause Generation

Brian Kell[1], Ashish Sabharwal[2], and Willem-Jan van Hoeve[3]

[1] Dept. of Mathematical Sciences, Carnegie Mellon University, Pittsburgh, PA 15213
bkell@cmu.edu
[2] Allen Institute for Artificial Intelligence, Seattle, WA 98103
AshishS@allenai.org
[3] Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213
vanhoeve@andrew.cmu.edu

Abstract. Nogood learning is a critical component of Boolean satisfiability (SAT) solvers, and increasingly popular in the context of integer programming and constraint programming. We present a generic method to learn valid clauses from exact or approximate binary decision diagrams (BDDs) and resolution in the context of SAT solving. We show that any clause learned from SAT conflict analysis can also be generated using our method, while, in addition, we can generate stronger clauses that cannot be derived from one application of conflict analysis. Importantly, since SAT instances are often too large for an exact BDD representation, we focus on BDD relaxations of polynomial size and show how they can still be used to generated useful clauses. Our experimental results show that when this method is used as a preprocessing step and the generated clauses are appended to the original instance, the size of the search tree for a SAT solver can be significantly reduced.

Introduction

Solvers for Boolean satisfiability (SAT) have become increasingly powerful in recent decades and can now be used to solve large-scale instances involving millions of variables and constraints. Much of the success of modern SAT solvers stems from their ability to quickly learn new constraints from infeasible search states via conflict-directed clause learning (CDCL). Conflict analysis has also been applied in the context of mixed-integer programming (MIP) [1, 17] and constraint programming (CP) [15, 21, 24] as "nogood" learning. In the context of constraint programming, nogood learning techniques have been proposed for specific combinatorial structures that arise from global constraints. For example, Downing et al. [12] study nogoods for global constraints that can be represented as a network flow. However, it remains a challenge to learn effective nogoods for MIP and CP solvers in a more generic context.

In this paper we introduce a generic approach for learning nogoods from decision diagrams, both exact and approximate. Decision diagrams provide a compact representation of the solution space for discrete optimization problems, and have been used to improve constraint propagation [2, 11, 14] and to derive optimization bounds [6, 8]. This work proposes an extension of the use of such

decision diagrams to learn nogoods. We specifically focus on clause learning in the context of SAT solving, being perhaps the most general form of nogood learning.

The architecture of today's SAT solvers, combining unit propagation with rapid restarts and CDCL, focuses on techniques with very low overhead and maximizes the number of search nodes that can be processed per second. While this has clearly been beneficial, the unit propagation inference performed by SAT solvers is arguably limited in strength. We therefore investigate a way to generate clauses that are *stronger* than those currently derived from unit propagation and CDCL. We show that these clauses, when added to the original formula, can substantially reduce the search tree size.

Our clause generation scheme is based on a novel application of binary decision diagrams (BDDs) to represent a given propositional formula. In contrast to conventional BDD construction methods that are context-agnostic, we associate a *meaning* with each node of the BDD: the set of clauses that are not yet satisfied. This allows us to apply a top-down compilation scheme [8] in which node equivalence is defined by the set of unsatisfied clauses. This node information provides a sufficient condition for efficiently creating BDDs (that are not necessarily reduced, however).

The key observation in our work is that the BDD node information, for those nodes that do not lead to a satisfying solution, can also be used to generate new clauses. Such clauses can be viewed as "nogoods" that forbid the solver to visit the associated search states. Since a node in a BDD can represent multiple partial assignments, a single nogood generated in this way is as strong as multiple nogoods derived from these separate partial assignments.

We formally characterize the strength of the clauses generated by our method. For example, we show that our clauses can indeed be stronger than one invocation of traditional conflict analysis. We also show the equivalence of our approach to regular and ordered resolution, which are specific restricted forms of resolution proofs.

BDDs that exactly represent a given CNF formula are well known to grow exponentially large in general. This has significantly limited the success of BDD-based techniques for SAT solving. To circumvent this limitation, we explore ways to apply our method to *relaxed* and *restricted* BDDs that represent a superset and subset, respectively, of all solutions instead [7, 8]. These approximate BDDs are created by merging non-equivalent nodes so as to respect a given limit on the size of the BDD. We show that the clauses derived from relaxed (or restricted) BDDs are still valid and can be computed efficiently.

We report results of computational experiments performed to evaluate the strength of our generated clauses in practice. We show that, for certain problem classes, our clauses can reduce the search tree size considerably. Interestingly, the solving time is not always reduced accordingly; we attribute this behavior to the length and number of our generated clauses. Nonetheless, the qualitative strength of our clauses demonstrates a great potential for inclusion in SAT solvers, and we propose several suggestions for doing so in the conclusion.

Binary Decision Diagrams

A *binary decision diagram* (BDD) [10, 18, 19, 25] is an edge-labeled acyclic directed multigraph whose nodes are arranged in $n+1$ layers $L_1, \ldots, L_{n+1}$. The layer L_1 consists of a single node, called the *root*. In this paper, every edge in the BDD is directed from a node in layer L_i to a node in layer L_{i+1}. Each node in layers $L_1, \ldots, L_n$ has two outgoing edges, one labeled "true" and the other labeled "false." There are two nodes in layer L_{n+1}, called the *sinks* or *terminals*; one of them, labeled $\top$, is the *true sink*, while the other, labeled $\bot$, is the *false sink*.

A BDD represents a Boolean function f defined on variables $x_1, \ldots, x_n$ as follows. The layers $L_1, \ldots, L_n$ correspond respectively to the variables $x_1, \ldots, x_n$. A path from the root to a sink corresponds to values of these variables; a "true" edge from a node in layer L_i to a node in layer L_{i+1} corresponds to $x_i = 1$, while a "false" edge corresponds to $x_i = 0$. If the path corresponding to the values of $x_1, \ldots, x_n$ ends at the true sink, then $f(x_1, \ldots, x_n) = 1$; otherwise the path ends at the false sink, and $f(x_1, \ldots, x_n) = 0$.

BDDs for SAT

An instance of the Boolean satisfiability (SAT) problem is a propositional formula on variables $x_1, \ldots, x_n$, expressed in conjunctive normal form (CNF), that is, as a conjunction of disjunctions of *literals*, where a literal is a variable x_i or its negation $\overline{x}_i$. Each of these disjunctions is called a *clause*. Because logical conjunction and disjunction are commutative, associative, and idempotent, we may view a SAT instance as a set of clauses, each of which is a set of literals. The objective is to determine whether there exists an assignment of Boolean values to the variables that simultaneously satisfies every clause.

Let I be a SAT instance on the variables $x_1, \ldots, x_n$, and let $\mathcal{S}$ denote the set of satisfying assignments to these variables. Let B be a BDD defined on the variables $x_1, \ldots, x_n$, and let $\mathcal{B}$ denote the set of assignments to these variables represented by B (that is, for which the Boolean function defined by B is true). If $\mathcal{B} = \mathcal{S}$, $\mathcal{B} \supseteq \mathcal{S}$, or $\mathcal{B} \subseteq \mathcal{S}$, then B is said to be an *exact BDD*, a *relaxed BDD*, or a *restricted BDD* for I, respectively [2, 7, 8, 13].

A path in a BDD from the root to a node in the layer L_{i+1} represents a *partial assignment*, i.e., an assignment $y \in \{0,1\}^i$ of values to the variables $x_1, \ldots, x_i$. Let $\mathcal{S}(y)$ denote the set of *satisfying completions* of this partial assignment, that is, $\mathcal{S}(y) = \{ z \in \{0,1\}^{n-i} : (y,z) \text{ is feasible} \}$. If y and y' are partial assignments with $\mathcal{S}(y) = \mathcal{S}(y')$, then we say that y and y' are *equivalent*. Note that in an exact BDD all paths from the root to a fixed node v represent equivalent partial assignments, and conversely if two partial assignments y and y' are equivalent then the paths in an exact BDD that correspond to y and y' can lead to the same node.

In the literature, BDDs are commonly required to be *reduced*, in the sense that any two equivalent partial assignments must be represented by the same node. The BDDs in this paper are not necessarily reduced.

In general, determining whether two partial assignments are equivalent is NP-hard for the SAT problem. However, we can sometimes determine that two partial assignments are equivalent by associating partial assignments with "states" [14, 16]. A *state function* for layer i is a map σ_i from the set $\{0,1\}^{i-1}$ of partial assignments at layer i into some set S_i of *states*, such that $\sigma_i(y) = \sigma_i(y')$ implies $\mathcal{S}(y) = \mathcal{S}(y')$. In other words, two partial assignments that lead to the same state have the same set of satisfying completions.

Behle [5] described a top-down algorithm for the construction of threshold BDDs, which are exact representations of solution sets of instances of 0–1 knapsack problems. A general algorithm for a top-down, layer-by-layer construction of a multivalued decision diagram (MDD), which is similar to a BDD except that the labels of the edges may come from any set, was given by Bergman et al. [8]. This algorithm works by maintaining state information for each node, computing the resulting state for each outgoing edge, and reusing nodes (i.e., pointing two edges at the same node) when the resulting states are the same.

To apply this top-down algorithm for the construction of a BDD from a SAT instance, we define $\sigma_i(y)$ for a partial assignment $y = \{y_1, \ldots, y_{i-1}\}$ to be the set of clauses in the instance that are not satisfied by the assignments $x_1 = y_1, \ldots, x_{i-1} = y_{i-1}$. Observe that if two partial assignments at layer i have the same set of unsatisfied clauses, then they have the same set of feasible completions, so this is indeed a state function. The state of the root node is the full set of clauses in the instance, and the state of a child node is formed from the state of its parent by removing all clauses that are satisfied by the variable assignment corresponding to the edge from the parent to the child.

Example 1. Consider a graph coloring problem on a complete graph with three vertices. Vertices 1 and 2 can be colored 0 or 1, while vertex 3 can be colored 0, 1, or 2. All nodes must be colored differently. We introduce variable x_1 for vertex 1, where $\overline{x}_1$ represents color 0 and x_1 represents color 1. Likewise we introduce x_2 for vertex 2. For vertex 3, we introduce three variables x_3, x_4, and x_5 for colors 0, 1, and 2, respectively. Here a positive literal represents that we choose that color, while its negation represents that we do not choose that color (e.g., $\overline{x}_3$ means that vertex 3 is not colored 0). We can formulate this problem as the following SAT instance with 11 clauses:

(1) $x_3 \vee x_4 \vee x_5$
(2) $\overline{x}_3 \vee \overline{x}_4 \vee \overline{x}_5$
(3) $\overline{x}_3 \vee \overline{x}_4$
(4) $\overline{x}_3 \vee \overline{x}_5$
(5) $\overline{x}_4 \vee \overline{x}_5$
(6) $x_1 \vee x_2$
(7) $\overline{x}_1 \vee \overline{x}_2$
(8) $x_1 \vee x_4 \vee x_5$
(9) $\overline{x}_1 \vee x_3 \vee x_5$
(10) $x_2 \vee x_4 \vee x_5$
(11) $\overline{x}_2 \vee x_3 \vee x_5$

The constructed BDD, using the lexicographic variable ordering, is presented in Figure 1. The state of each node is the set of (indices of) clauses that have not been satisfied by any path from the root to that node. "True" edges are drawn as solid lines, and "false" edges are drawn as dashed lines. Infeasible nodes, that is, nodes from which no path leads to the true sink, are shaded gray.

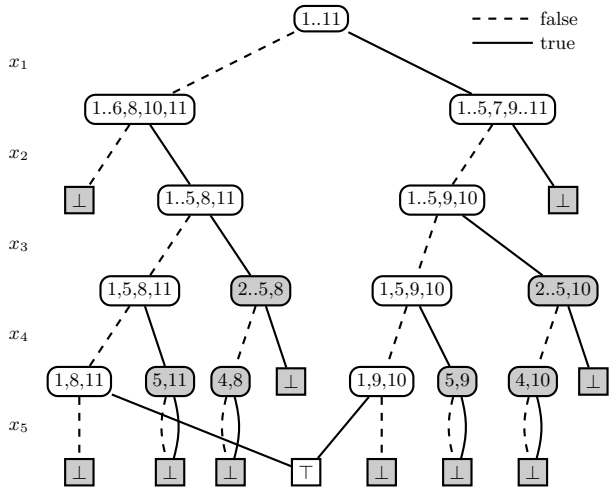

Fig. 1. The exact BDD for the example. The false sink is drawn multiple times for clarity.

Approximate BDDs

In general, exact BDDs can be of exponential size, so the construction of an exact BDD may not be practical. For this reason, it is useful to consider BDDs that represent relaxations or restrictions of the SAT instance. Such BDDs are called *approximate BDDs* because their structure approximates the structure of the exact BDD.

MDDs of limited width were proposed by Andersen et al. [2] to reduce space requirements. In this approach, the MDD is constructed in a top-down, layer-by-layer manner, and whenever a layer of the MDD exceeds some predetermined value W an approximation operation is applied to reduce its size to W before constructing the next layer. One way to perform this approximation is to use a relaxation (or restriction) operation $\oplus$ defined on the states of nodes so that, given nodes v and v', the state given by $\text{state}(v) \oplus \text{state}(v')$ is a "relaxation" (or "restriction") of both $\text{state}(v)$ and $\text{state}(v')$. A subset of nodes in a layer can be merged into a single node by applying this operator to obtain a state for the new node, and this is repeated until the size of the layer is reduced to W [8, 14, 16].

In our case, the state of a node is a set of unsatisfied clauses, and so the appropriate relaxation operation is the intersection operation (and the appropriate restriction operation is the union operation).

Clause Generation with BDDs

We propose the use of a BDD representation of a SAT instance to generate clauses. One simple way to deduce clauses from a BDD is to project the variable

assignments along the satisfying paths in a BDD and to look for variables whose values must be fixed. For instance, in the example above, we can infer from both feasible paths that we can fix $\overline{x}_3$, $\overline{x}_4$, and x_5. However, in practice we must use approximate BDDs, and this approach does not produce much useful information.

A more fruitful approach is to deduce clauses from the infeasible nodes of the BDD (that is, the nodes from which no path leads to the true sink) by using the state information for these nodes. In particular, we generate a clause for each infeasible node in the BDD that witnesses its infeasibility. We do this systematically by applying a sequence of resolution steps.

Resolution is a commonly used inference rule applied to propositional formulas in conjunctive normal form. The resolution rule, applied to two clauses $x_i \vee P$ and $\overline{x}_i \vee Q$, where P and Q denote disjunctions of literals, is

$$\frac{x_i \vee P \qquad \overline{x}_i \vee Q}{P \vee Q}.$$

The resulting clause $P \vee Q$ is called the *resolvent*.

During the top-down construction of a BDD for a SAT instance, infeasibility of a state is detected when an unsatisfied clause contains no variable corresponding to a lower layer of the BDD. When this occurs, we choose one such clause as a witness of the infeasibility of the corresponding node.

After the BDD construction is complete, we perform a single bottom-up pass to identify all infeasible nodes and generate a witness clause for each. For an infeasible node v in layer L_i, we generate a witness clause as follows:

- If one of the child states has a witness clause that does not contain the variable x_i, then choose this clause as the witness clause for v.
- Otherwise, one child has a witness clause containing x_i and the other has a witness clause containing $\overline{x}_i$, so apply the resolution rule to these two clauses with respect to the variable x_i and use the resolvent as the witness clause for v.

At the end, we output the witness clauses for all roots of maximal infeasible subtrees of the BDD.

Example 2. Continuing the graph-coloring example from earlier, consider the infeasible subtree rooted at the node with state $\{2, 3, 4, 5, 8\}$ in layer L_4 in Figure 1. This subtree is redrawn in Figure 2. Setting $x_4 = 0$ satisfies clauses 2, 3, and 5, so the "false" child (i.e., the child along the "false" edge) has state $\{4, 8\}$. However, from this node, setting $x_5 = 0$ means that clause 8 cannot be satisfied, and setting $x_5 = 1$ means that clause 4 cannot be satisfied. Therefore, neither of the children of the node with state $\{4, 8\}$ is feasible, and we have a witness of the infeasibility of each: clause 8, $x_1 \vee x_4 \vee x_5$, for the "false" child, and clause 4, $\overline{x}_3 \vee \overline{x}_5$, for the "true" child.

Likewise, returning to the node with state $\{2, 3, 4, 5, 8\}$, if we set $x_4 = 1$ then clause 3 cannot be satisfied, so clause 3, $\overline{x}_3 \vee \overline{x}_4$, is a witness of the infeasibility of this child.

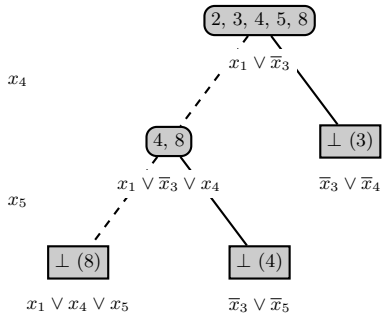

Fig. 2. Witness clauses generated from the infeasible subtree rooted at the node with state $\{2, 3, 4, 5, 8\}$ in layer L_4.

Now, in our bottom-up pass, we first determine that the node with state $\{4, 8\}$ is infeasible. Both of its child nodes have witness clauses that contain the variable x_5, so we apply the resolution rule to these two witness clauses with respect to x_5 to obtain the clause $x_1 \vee \overline{x}_3 \vee x_4$, which is a witness of the infeasibility of the node with state $\{4, 8\}$. Likewise, the node with state $\{2, 3, 4, 5, 8\}$ is infeasible, so we apply the resolution rule to these two witness clauses with respect to x_4 to obtain the clause $x_1 \vee \overline{x}_3$.

Since the node with state $\{2, 3, 4, 5, 8\}$ is the root of a maximal infeasible subtree of the BDD, we produce the clause $x_1 \vee \overline{x}_3$ as output. This is a valid clause for the original SAT instance.

In a similar way, we generate the witness clause $x_2 \vee \overline{x}_3$ for the node with state $\{2, 3, 4, 5, 10\}$ in layer L_4 in the BDD in Figure 1.

Characterization of Generated Clauses

Let us formally define a clause C to be *valid* for a propositional formula F if $F \models C$, i.e., F logically entails C. In other words, $F \wedge C$ has the same set of solutions as F itself. We begin with a few properties of witness clauses generated using the BDD method.

Theorem 1. *Let F be a CNF formula and B be a top-down exact, relaxed, or restricted BDD for F constructed as described above. Then:*

1. *Every witness clause generated from B is valid for F.*
2. *The set of variables in every witness clause generated at layer L_{i+1} is a subset of $\{x_1, x_2, \ldots, x_i\}$.*
3. *If B is an exact or relaxed BDD, the witness clause C generated for a node v of B is falsified by the partial assignment corresponding to every path from the root of B to v. In particular, C does not contain any variable that appears both negatively and positively in paths from the root to v.*
4. *If B is an exact or relaxed BDD, the witness clause C associated with any infeasible node v of B witnesses the infeasibility of v.*

5. *Let U denote the roots of maximal infeasible subtrees of B. If B is exact, then the set G of all witness clauses associated with nodes $v \in U$ is a reformulation of F.*

Proof. The first claim follows immediately from the observation that the witness clause C associated with any node v is derived using a sequence of resolution operations starting from the clauses in state(v). Since resolution is a sound proof system, state(v), and hence F, must entail C.

We prove the second claim by induction on i. For $i = n+1$, the claim trivially holds. Suppose the claim holds for clauses generated at layer L_i, with $i > 1$. By construction, any clause C generated at layer L_{i-1} either is identical to a clause generated at layer L_i, in which case it does not contain the variable x_{i-1}, or else is obtained by resolving two clauses at layer L_i on the variable x_i. In either case, by the induction hypothesis, the variables appearing in C must be a subset of $\{x_1, x_2, \ldots, x_{i-2}\}$.

To prove the third claim, we recall from the definition of the state function that when B is exact or relaxed, the partial assignment y corresponding to any path from the root to v does not satisfy any clause in state(v) $= F_v \subseteq F$. For the sake of contradiction, suppose ℓ is a literal of the witness clause C that is satisfied by y. Since C is derived by applying resolution steps to clauses in F_v, the literal ℓ must appear in at least one clause C' of F_v. Since y satisfies ℓ, it would also satisfy C', a contradiction. Hence, C must be falsified by y. Finally, if C contained a literal ℓ that appears positively and negatively in partial assignments y and y' corresponding to two paths from root to v, then C would clearly be satisfied by at least one of y and y', which, as proved above, cannot happen. Hence, C must not contain any such literal.

For proving the fourth claim, we use the above property that when B is exact or relaxed, the partial assignment y corresponding to any path from the root to v does not satisfy C. Suppose y could be extended to a full assignment (y, z) that satisfies F. Then z must satisfy all clauses in state(v) $= F_v$ as these clauses, by definition of the state function, are not satisfied by y. Since C is derived from F_v by applying a sequence of resolution operations, z must then also satisfy C. However, as observed above, C is a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$, where L_i is the layer containing v, and hence C cannot possibly be satisfied by z. This proves that y cannot be extended to a full assignment satisfying F, and that the generated clause C witnesses this fact as well as the infeasibility of v.

Lastly, when B is exact, we argue that the set G of witness clauses associated with roots of maximal infeasible subtrees of B is logically equivalent to F. If y is a solution to F, then y must satisfy all witness clauses as these clauses are entailed by F. Hence y must also satisfy G. On the other hand, if y is not a solution to F, then let y' be the partial assignment corresponding to the path in B associated with y but truncated at the root v' of a maximal infeasible subtree. By the third property above, y' (and hence y) must falsify the clause C' associated with v', and hence falsify G. It follows that F and G have the same set of solutions and thus G is a reformulation of F. □

In the remainder of this section, we explore how BDD-guided clause generation relates to propagation and inference techniques used in today's SAT solvers. To make this connection precise, we recall the notion of *absorbed clauses* [3, 22]. A clause C is said to be absorbed by a CNF formula F if for every literal $\ell \in C$, performing unit propagation[4] on F starting with all literals of C except ℓ set to false either infers ℓ or infers a conflict. The intuition here is that C is absorbed by F if F and $F \wedge C$ have identical entailment power with respect to unit propagation, i.e., whatever one can derive from $F \wedge C$ using unit propagation one can also derive from F itself.

Pipatsrisawat and Darwiche [22] showed that the conflict-directed clause learning (CDCL) mechanism in SAT solvers always produces clauses that are not absorbed by the current theory, that is, by the set of initial clauses of F and those learned thus far during the search. As we show next, this property also holds for clauses generated by the BDD method applied to F, as long as we ignore any states at which unit propagation already identifies a conflict.

More formally, given a BDD B, let us define a *unit-propagated BDD*, denoted B_{up}, as the one obtained by removing from B all nodes v such that unit propagation on state(v) results in a conflict. From a practical standpoint, such nodes can be easily identified in linear time and discarded.

Theorem 2. *Let F be a CNF formula and B be a top-down exact, relaxed, or restricted BDD for F constructed as described above. Let C be the witness clause for the root v of any maximal infeasible subtree in B_{up}. Then C is not absorbed by F.*

Proof. We first show that setting all literals of C to false and performing unit propagation does not result in a conflict. From Theorem 1, the partial assignment y corresponding to the path from the root of B_{up} to y falsifies C. Yet, by design, unit propagation on state(v) does not result in a conflict. Therefore, it must be the case that unit propagation on F starting with the partial assignment y does not result in a conflict. Since y falsifies all literals of C, we infer that falsifying all literals of C and performing unit propagation does not result in a conflict.

To finish the argument that C is not absorbed by F, we next show that there exists a literal ℓ in C such that setting all literals of C except ℓ to false does not allow unit propagation to infer ℓ. Since v is the root of a maximal infeasible subtree in B_{up}, it must be the case that v has a sibling node v' that is *not* identified as being infeasible. Let u be the common parent of v and v'. The witness clause C associated with v must include a literal ℓ corresponding to the branching variable associated with the layer of u. We will use ℓ to demonstrate that C is not absorbed by F.

Suppose, for the sake of contradiction, that setting all literals of C other than ℓ to false and performing unit propagation infers ℓ. Consider the partial

[4] *Unit propagation* on a CNF formula is the process of identifying, if there is one, a clause that contains only one literal ℓ, setting ℓ to true, simplifying the formula by removing $\bar{\ell}$ from all clauses and removing all clauses containing ℓ, and repeating. Unit propagation is said to result in a *conflict* if it generates an empty clause.

assignment z corresponding to the path from the root of B_{up} to u. This partial assignment z differs from the partial assignment y identified above in only the literal ℓ. It must then be the case that z falsifies all literals of C except for ℓ, and therefore unit propagation on state(u) starting with z must infer ℓ. In other words, there exists a clause C' in state(u) such that $\ell \in C'$ and z, after unit propagation, falsifies all literals of C' other than ℓ. This, however, implies that C' is also in state(v'), the state corresponding to the sibling v' of v, and further that unit propagation on state(v') must falsify C', resulting in a conflict. This, however, contradicts the fact that v' was not identified as an infeasible node in B_{up}. □

This establishes that our clause generation approach effectively produces clauses that provide useful information not already captured by unit propagation inference on F.

While a series of potentially exponentially many applications of the CDCL mechanism can eventually let the solver learn any clause entailed by F (including the empty clause in case F is unsatisfiable), we show below that any clause that it can learn with *one* application of conflict analysis starting from a clause set F is a special case of the BDD-generated clauses starting from F. This holds for any clause learning scheme employed by the solver to choose a cut in the underlying conflict graph.[5]

The proof of this claim uses properties of a few different restrictions of general resolution which we briefly recapitulate. A *tree-like resolution* is one where no clause, other than the initial clauses of F, is used in more than one resolution step. A *regular resolution* is one where no variable is resolved upon more than once in any root-to-leaf path. Finally, an *ordered resolution* is one where the order of variables resolved upon is identical across all root-to-leaf paths.

Theorem 3. *For any clause C learned from one application of SAT conflict analysis on F using any clause learning scheme, there exists a variable ordering under which a top-down approximate BDD of width at most $2^{|C|}$ for F generates a clause $C' \subseteq C$.*

Proof. To prove this, we use the resolution-based characterization of CDCL clauses [4], namely, the CDCL derivation of a clause C starting from F and using any clause learning scheme can be viewed as a very simple form of resolution derivation that has a ladder-like structure. More formally, the derivation τ of C is simultaneously a tree-like, regular, linear, and ordered resolution derivation from the clauses in F. This means that each intermediate clause C_{j+1} in τ is obtained by resolving C_j with a clause of F and that the sequence σ of variables resolved upon in τ consists of all distinct variables.

We can use BDDs to derive from F a clause C' that, together with F, absorbs C. To construct such a BDD B, we use as the top-down (partial) variable order first the variables that appear in C (in any order) followed by variables

[5] The specifics of the SAT conflict analysis terminology are not critical here. The interested reader is referred to relevant surveys such as by Marques-Silva et al. [20].

in the reverse order of σ. The first $|C|$ variables result in a BDD of width at most $2^{|C|}$. Let v be the node of B in the layer $L_{|C|+1}$ at which all literals of C are falsified. When expanding B from v, the ladder-like structure of τ guarantees that at least one branch on the variables in σ can be labeled directly by a clause of F that is falsified. The corresponding lower part of B starting at v is thus of width 1. For the remaining $2^{|C|} - 1$ nodes of B in the layer $L_{|C|+1}$, we construct an approximate lower portion of the BDD such that the overall width does not increase. This makes the overall width of B be $2^{|C|}$.

While B may have several infeasible nodes, the node v in the layer $L_{|C|+1}$ is guaranteed by the derivation τ to be infeasible. Recall that the path p from the root of B to v falsifies C. Consider the node v' that is the root of the maximal infeasible subtree of B that contains v. Let C' be the BDD-generated clause witnessing the infeasibility of v'. By Theorem 1, C' must be falsified by the path p' from the root of B to v'. Note that p' is a sub-path of p. By construction, C contains all $|C|$ literals mentioned along p, while, by Theorem 1, C' contains a subset of the literals mentioned along p' and hence along p. It follows that $C' \subseteq C$. □

The above reasoning can be extended to construct an *exact* BDD that generates a subclause of C. However, the width of such a BDD will depend not only on $|C|$ but also on the number of resolution steps involved in conflict analysis during the derivation of C.

Theorem 4. *Clauses generated by applying the BDD method to F correspond to regular and ordered resolution derivations starting from the clauses of F.*

Proof. It is easily seen that the resolution operations performed during clause generation from a BDD respect, by construction, the restrictions of being regular and ordered. Hence, any BDD generated clause C can be derived using regular and ordered resolution starting from F.

On the other hand, let τ be any regular and ordered resolution derivation of C starting from F. An argument similar to the one in the proof of Theorem 3 can be used to show that there exists a natural variable order (namely, first branch on the variables of C, then follow the top-down variable order imposed by τ) under which the top-down BDD B for F contains a node v such that the path from the root of B to v falsifies all literals of C. As before, witness clauses for B may not directly include C as is, but the witness clause C' associated with the root of the maximal infeasible subtree of B containing v would be a subclause of C. □

We recall again the resolution-based characterization of CDCL clauses, namely, those that can be derived using tree-like, regular, linear, and ordered resolution. This results in linear-size resolution derivations and thus forms a strict subset of all possible derivations that are regular and ordered, but not necessarily tree-like and linear. The above theorem therefore implies the following:

Corollary 1. *There exist BDD-generated clauses that cannot be derived using one application of SAT conflict analysis.*

Implementation and Experimental Results

We implemented the clause generation algorithm described above in C++, as a program called Clausegen. Several implementation decisions needed to be considered.

The variable ordering used in a BDD can have a very significant effect on the size of the BDD (and consequently the quality of an approximate BDD). Unfortunately, determining the optimal variable ordering is very difficult; in general, the problem of determining whether a given variable ordering of a BDD can be improved is NP-complete [9]. For our implementation, we use a simple heuristic to determine the variable ordering: each variable is assigned a score, computed as the quotient between the number of clauses containing the variable and the average arity of those clauses, and the variables are sorted in decreasing order according to this score, so that higher-scoring variables (that is, variables that appear in many mostly short clauses) correspond to layers nearer the top of the BDD.

The construction of a relaxed BDD via merging also requires a rule for determining which nodes to merge in a layer that exceeds the maximum width. Since unsatisfied clauses lead to infeasibility, and our method generates clauses from infeasible subtrees, the following merging rule is used: if a constructed layer exceeds the maximum width W, sort the nodes by the number of unsatisfied clauses in their states, preserve the $W - 1$ nodes with the greatest number of unsatisfied clauses, and merge the other nodes into a single node. (The state of the resulting node is the intersection of the states of the nodes that were merged.) Merging rules similar to this one have been applied before in the context of optimization and scheduling, for example by Cire and van Hoeve [11].

To demonstrate our method, we considered SAT instances produced from randomly generated bipartite graph matching problems, with 15 vertices on each side, in which a random subset of 10 vertices on one side is matched with only 9 vertices on the other side, so that the graph fails to satisfy Hall's condition, thereby making the SAT instance unsatisfiable. We preprocessed the instance with SatELite 1.0 (using the `+pre` option) and used Minisat 2.2.0 as the SAT solver (with `-rnd-freq=0.01`). Because Minisat uses a nondeterministic algorithm, it was run 20 times for each test with different random seeds, and the results were averaged. The experiments were run on an Intel Xeon E5345 at 2.33 GHz with 24 GB of RAM running Ubuntu 12.04.5.

For a representative instance of this type, with 225 variables and 748 clauses (80 variables and 405 clauses after preprocessing), Minisat made 864,930 decisions and encountered 714,625 conflicts on average.

Figure 3 shows the results of appending the clauses produced by Clausegen before the instance is given to Minisat. As the maximum BDD width is increased from 10 to 10,000, thus yielding more accurate approximate BDDs, the numbers of decisions and conflicts encountered by Minisat decrease. The clauses generated at BDD width 10,000 produced an improvement in these metrics by over 75% in comparison with the original instance: Minisat averaged 212,158 decisions and 178,101 conflicts.

However, we do not see a corresponding improvement in the running time of Minisat. The stacked area plot in Figure 3 shows the running time of Clausegen and Minisat as the BDD width is increased. On the original instance, Minisat required an average of 7.83 s; this time increased to 17.65 s when the clauses generated at BDD width 10,000 were added. The number of generated clauses increases linearly with the BDD width, from 12 clauses at width 10 to 9745 clauses at width 10,000. The clauses generated at width 10,000 have an average length of 11.8, compared to an average length of 2.1 in the original instance.

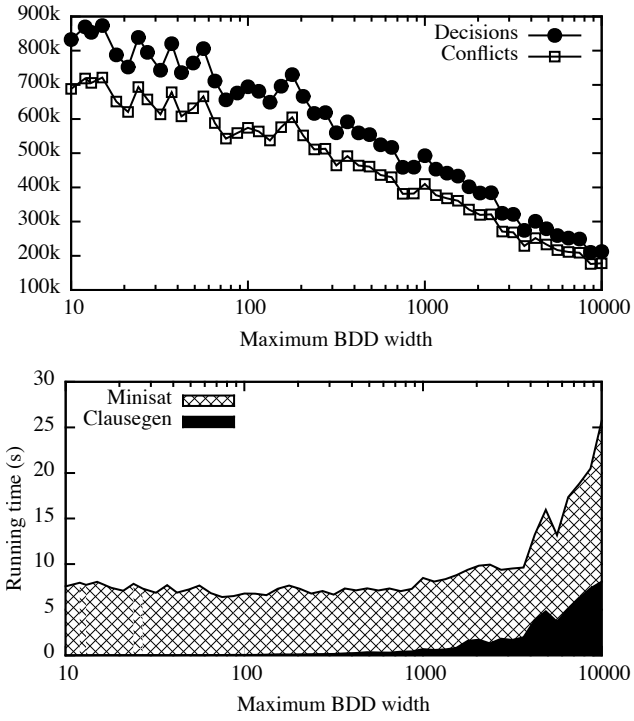

Fig. 3. Minisat statistics for an unsatisfiable bipartite matching instance.

Figure 4 shows our results for another instance, counting-clqcolor-unsat-set-b-clqcolor-08-06-07.sat05-1257.reshuffled-07.cnf, from the SAT Challenge 2012 Hard Combinatorial SAT+UNSAT benchmark instances [23]. This instance has 132 variables and 1527 clauses of average length 2.9 (117 variables and 1599 clauses of average length 4.3 after preprocessing with SatELite) and is also unsatisfiable; it represents a graph coloring instance with a hidden clique that is larger than the number of colors available. Minisat averaged 2,072,107 decisions and 1,511,029 conflicts for the original instance, taking 14.18 s on average. When the 3255 clauses of average length 8.7 produced by Clausegen at BDD

width 10,000 were added, the average numbers of decisions and conflicts decreased to 713,718 and 515,514, respectively, and the average running time of Minisat decreased to 6.34 s. The minimum total running time of Clausegen and Minisat together was achieved at a BDD width of 464; Clausegen took 0.57 s to generate 340 clauses of average length 8.7, and Minisat averaged 1,351,691 decisions and 972,674 conflicts, taking 9.14 s on average to solve the instance, for a total average solving time of 9.71 s (an improvement of 31.5% over the original instance).

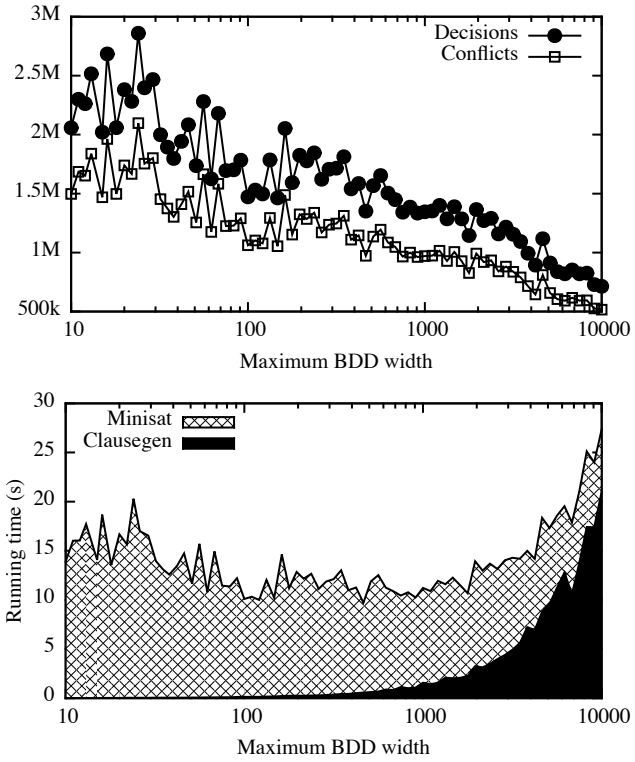

Fig. 4. Minisat statistics for counting-clqcolor-unsat-[. . .].cnf from SAT Challenge 2012.

It may be that not all of the generated clauses are necessary. Preliminary experiments involving the selection of random subsets of the generated clauses appear to indicate that some subsets are significantly more helpful than others. A heuristic to select useful subsets of the generated clauses may allow a decrease in running time to match the decreases in the numbers of decisions and conflicts.

Conclusion

We presented a new algorithm that uses BDDs and resolution to generate valid clauses from a SAT instance. This algorithm can use approximate BDDs for instances that are too large for an exact BDD. We compared the strength of our method to that of SAT conflict analysis and showed that our method can generate strictly stronger clauses than a single application of SAT conflict analysis. Our experimental results show that concatenating these generated clauses to the original instance can significantly reduce the size of the search tree for a SAT solver.

For a practical implementation of our method, we propose the following techniques to improve computational efficiency. First, initial experimentation has shown that not all generated clauses are equally effective. We therefore suggest the development of a heuristic to select and add only a small subset. Second, a large formula may be decomposable into subformulas that are each representable effectively by a BDD. It seems natural to make such a decomposition based on structural properties of the formula (e.g., properties of the constraint graph). Third, when a SAT solver appears to be making very little progress, and the number of remaining free variables is limited, we can interrupt the search and give the rest of the formula to a BDD to generate clauses conditional on the partial assignment represented by the last search state.

References

[1] T. Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, 2007.

[2] H. Andersen, T. Hadzic, J. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. *Principles and Practice of Constraint Programming–CP 2007*, pp. 118–132, 2007.

[3] A. Atserias, J. K. Fichte, and M. Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373, 2011.

[4] P. Beame, H. Kautz, and A. Sabharwal. Understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22: 319–351, Dec. 2004.

[5] M. Behle. On threshold BDDs and the optimal variable ordering problem. *Journal of Combinatorial Optimization*, 16:107–118, 2008. ISSN 1382-6905.

[6] B. Bergman, A. Cire, W.-J. van Hoeve, and J. Hooker. Optimization Bounds from Binary Decision Diagrams. *INFORMS Journal on Computing*, 26(2): 253–268, 2014.

[7] D. Bergman, A. A. Cire, W.-J. van Hoeve, and T. Yunes. BDD-based heuristics for binary optimization. *Journal of Heuristics*, 20(2):211–234, 2014. ISSN 1381-1231.

[8] D. Bergman, W.-J. van Hoeve, and J. Hooker. Manipulating MDD relaxations for combinatorial optimization. In T. Achterberg and J. Beck,

editors, *Proceedings of the 8th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, Lecture Notes in Computer Science 6697, pp. 20–35, Berlin, May 2011. Springer.

[9] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *Computers, IEEE Transactions on*, 45(9):993–1002, 1996.

[10] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, Aug. 1986. ISSN 0018-9340.

[11] A. Cire and W.-J. van Hoeve. Multivalued Decision Diagrams for Sequencing Problems. *Operations Research*, 61(6):1411–1428, 2013.

[12] N. Downing, T. Feydy, and P. Stuckey. Explaining Flow-Based Propagation. In *Proceedings of CPAIOR*, pp. 146–162, 2012.

[13] T. Hadzic, J. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In *Principles and Practice of Constraint Programming*, pp. 448–462. Springer, 2008.

[14] S. Hoda, W.-J. van Hoeve, and J. Hooker. A systematic approach to MDD-based constraint programming. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, CP'10, pp. 266–280, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15395-X, 978-3-642-15395-2. URL `http://dl.acm.org/citation.cfm?id=1886008.1886034`.

[15] G. Katsirelos. *Nogood Processing in CSPs*. PhD thesis, University of Toronto, 2008.

[16] B. Kell and W.-J. van Hoeve. An MDD approach to multidimensional bin packing. In C. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, vol. 7874 of *Lecture Notes in Computer Science*, pp. 128–143. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38170-6.

[17] F. Kilinç-Karzan, G. Nemhauser, and M. Savelsbergh. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation*, 1(4):249–293, 2009.

[18] D. E. Knuth. *The Art of Computer Programming*, vol. 4, fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley, 2009.

[19] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959. ISSN 1538-7305.

[20] J. P. Marques-Silva, I. Lynce, and S. Malik. CDCL solvers. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, chapter 4, pp. 131–154. IOS Press, 2009.

[21] O. Ohrimenko, P. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14:357–391, 2009.

[22] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011.

[23] SAT Challenge 2012. SAT Challenge 2012: Downloads. `http://baldur.iti.kit.edu/SAT-Challenge-2012/downloads.html`, 2012. [Online; accessed November 23, 2014].

[24] P. Stuckey. Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving. In *Proceedings of CPAIOR*, pp. 5–9. Springer, 2010.

[25] I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, 2000.