Backbones for Equality

Michael Codish[1], Yoav Fekete[1], and Amit Metodi[2,⋆]

[1] Department of Computer Science, Ben-Gurion University, Israel
{mcodish,fekete}@cs.bgu.ac.il
[2] Cadence Israel Development Center, Israel
ametodi@cadence.com

Abstract. This paper generalizes the notion of the backbone of a CNF formula to capture also equations between literals. Each such equation applies to remove a variable from the original formula thus simplifying the formula without changing its satisfiability, or the number of its satisfying assignments. We prove that for a formula with n variables, the generalized backbone is computed with at most $n+1$ satisfiable calls and exactly one unsatisfiable call to the SAT solver. We illustrate the integration of generalized backbone computation to facilitate the encoding of finite domain constraints to SAT. In this context generalized backbones are computed for small groups of constraints and then propagated to simplify the entire constraint model. A preliminary experimental evaluation is provided.

1 Introduction

The backbone of a search problem is a fundamental notion identified to explain why certain problem instances are hard. The term originates in computational physics [8,17,16]. It identifies decisions which are fixed in all solutions, and so need to be made correctly. Typically, a decision is the value of a variable, and if that value is fixed in all solutions then the variable is called a backbone variable. If a problem has a backbone variable, an algorithm will not find a solution to the problem until the backbone variable is set to its correct value. Therefore, the larger a backbone, the more tightly constrained the problem becomes. As a result, it is more likely for an algorithm to set a backbone variable to a wrong value, which may consequently require a large amount of computation to recover from such a mistake [20].

For SAT, the backbone of a satisfiable propositional formula φ is the set of variables which take the same truth value in all satisfying assignments of φ. In this case, the backbone can also be seen as the set of literals which are true in all satisfying assignments of φ. Computing the backbone of a propositional formula is intractable in general [7]. Janota proves that deciding if a literal is in the backbone of a formula is co-NP [6] and Kilby *et al.* show that even approximating the backbone is intractable [7].

⋆ This research was carried out while the third author was a graduate student at Ben-Gurion University.

V. Bertacco and A. Legay (Eds.): HVC 2013, LNCS 8244, pp. 1–14, 2013.
© Springer International Publishing Switzerland 2013

Backbones appear in a number of practical applications of SAT. If a backbone is known, then we can simplify a formula without changing its satisfiability, or the number of satisfying assignments. Assigning values to backbone variables reduces the size of the search space while maintaining the meaning of the original formula. On the other hand, computing the backbone of a SAT problem is typically at least as hard as solving the SAT problem itself. Investing the cost of computing a backbone (or part of it) can pay off when the application must solve the same formula many times. Typical examples are model enumeration, minimal model computation, prime implicant computation, and also in applications which involve optimization (see for example, [12]). Another useful application, examplified in [10], is when SAT solving is incremental, and a backbone can be computed for a small portion of the CNF but used to simplify the whole CNF.

Backbones are often computed by iterating with a SAT solver. For a satisfiable propositional formula φ and literal x, if $\varphi \wedge \neg x$ is not satisfiable, then x is in the backbone. For a formula with n variables and a backbone consisting of b literals, a naive approach requires $2n$ calls to the SAT solver from which b are unsatisfiable and typically more expensive. In [12], the authors survey several less naive options and introduce an improved algorithm. For a formula with n variables and a backbone with b literals their algorithm requires at most $n - b$ satisfiable calls and exactly one unsatisfiable call to the SAT solver.

This paper generalizes the notion of the backbone of a CNF formula φ to capture all equations of the form $x = \ell$ implied by φ where x is a variable, and ℓ is either a truth value or a literal. In this case we say that x is a generalized backbone variable. The (usual) backbone of φ is the subset of these equations where ℓ is a truth value. The motivation for generalized backbones is exactly the same as for backbones: each implied equation represents a decision which is fixed in all solutions, and if we know that $x = \ell$ is implied by the formula then all occurrences of x can be replaced by ℓ thus fixing the decision and simplifying the formula without changing its satisfiability, or the number of its satisfying assignments.

We prove that generalized backbones (with equalities) are not much more expensive to compute than usual backbones. We show that for a formula with n variables the generalized backbone is computed with at most $n+1$ satisfiable calls and exactly one unsatisfiable call to the SAT solver. We also illustrate through preliminary experimentation that computing generalized backbones does pay off in practice.

In previous work described in [14,13,15], we take a structured approach to solve finite domain constraint problems through an encoding to SAT. With this approach we partition a CNF encoding into smaller chunks of clauses, determined by the structure of the constraint model, and we reason, one chunk at a time, to identify (generalized) backbone variables. Clearly, any (generalized) backbone variable (or an implied equation) of a single chunk is also a (generalized) backbone variable (or an implied equation) of the entire CNF. Moreover, a backbone variable identified in one chunk may apply to simplify other chunks. In [14], we termed the process of identifying such equations, and propagating them to other

chunks, equi-propagation. We introduced a tool called BEE (Ben-Gurion Equi-propagation Encoder) which applies to encode finite domain constraint models to CNF. During the encoding process, BEE performs optimizations based on equi-propagation and partial evaluation to improve the quality of the target CNF. However, equi-propagation in BEE is based on ad-hoc rules and thus incomplete.

In this paper we describe the extension of BEE to consider complete equi-propagation (CEP) which is about inferring generalized backbones for chunks of the CNF encoding and propagating them to simplify the entire CNF. In this setting, chunks of CNF designated for complete equi-propagation are specified by the user in terms of sets of constraints. For each such specified set, an algorithm for generalized backbone computation is applied to its CNF encoding. For typical constraint satisfaction problems, removing some of the constraints renders a CNF which is much easier to solve. Hence, here too, the cost of computing the (generalized) backbone of an individual chunk can pay off when applied in the global context to solve the whole CNF.

2 Related Work

Simplifying CNF formula prior to the application of SAT solving is of the utmost importance and there are a wide range of preprocessing techniques that can be applied to achieve this goal. See for example the works of [9], [2], [4], and [11], and the references therein their work. Detecting unit clauses and implications (and thus also equalities) between literals is a central theme in CNF preprocessing. The preprocessor described in [5] focuses on detecting precisely the same kind of equations we consider for generalized backbones: unit clauses and equalities between literals.

There are also approaches [9] that detect and use Boolean equalities during run-time, from within the SAT solver. Perhaps the most famous example is the SAT solver of Stålmark [19] which has extensive support for reasoning about equivalences and where formulae are represented in a form containing only conjunctions, equalities and negations [18].

The approach taken in this paper is different from these works. The above mentioned works apply various techniques (resolution based and others) to track down implications. They are not complete techniques. Ours is a preprocessing technique with a focus on the computation of complete equi-propagation implemented using a backbone algorithm (with equalities). A key factor is that by considering only a small fragment of a CNF at one time enables to apply stronger, and even complete, reasoning to detect generalized backbones in that fragment. Once detected, these apply to simplify the entire CNF and facilitate further reasoning on other fragments.

When compiling finite domain constraints to CNF using the BEE compiler, the structure of the constraints can be applied to induce a partition of the target CNF to such fragments.

3 Backbones and Equalities

In this section we first describe an algorithm for computing backbones and then detail its application to compute generalized backbones (with equality). Our approach is essentially the same as Algorithm 4 presented in [12].

To compute the backbone of a given formula φ, which we assume is satisfiable, we proceed as follows: the algorithm maintains a table indicating for each variable x in φ for which values of x, φ can be satisfied: *true*, *false*, or both. The algorithm is initialized by calling the SAT solver with $\varphi_1 = \varphi$ and initializing the table with the information relevant to each variable: if the solution for φ_1 assigns a value to x then that value is tabled for x. If it assigns no value to x then both values are tabled for x.

The algorithm iterates incrementally. For each step $i > 1$ we add a single clause C_i (detailed below) and re-invoke the same solver instance, maintaining the learned data of the solver. This process terminates with a single unsatisfiable invocation. In words: the clause C_i can be seen as asking the solver if it is possible to flip the value for any of the variables for which we have so far seen only a single value. More formally, at each step of the algorithm, C_i is defined as follows: for each variable x, if the table indicates a single value v for x then C_i includes $\neg v$. Otherwise, if the table indicates two values for x then there is no corresponding literal in C_i. The SAT solver is then called with $\varphi_i = \varphi_{i-1} \wedge C_i$. If this call is satisfiable then the table is updated to record new values for variables (there must be at least one new value in the table) and we iterate. Otherwise, the algorithm terminates and the variables remaining with single entries in the table are the backbone of φ.

In [12] the authors prove[1] that for a formula with n variables and a backbone with b literals the above algorithm requires at most $n - b$ satisfiable calls and exactly one unsatisfiable call to the SAT solver. The following example demonstrates the application of the backbone algorithm.

	$\overbrace{x_1\ x_2\ x_3\ x_4\ x_5}^{\theta_i}$	φ_i
i=1	1 1 0 0 1	φ
i=2	1 0 0 1 0	$\varphi_1 \wedge \neg\theta_1$
i=3	*unsat*	$\varphi_2 \wedge (\neg x_1 \vee x_3)$

Fig. 1. Demo of backbone algorithm (Example 1)

Example 1. Assume given an unspecified formula, φ, with 5 variables. Figure 1 illustrates the iterations (three in this example) of the backbone algorithm: one per line. The columns in the table detail the iteration number i (left), the the formula φ_i provided to the SAT solver (right), and the model θ_i obtained (middle). The example illustrates that $\varphi_1 = \varphi$ has a model θ_1 depicted in the first

[1] See Proposition 6 in `http://sat.inesc-id.pt/~mikolas/bb-aicom-preprint.pdf`

line of the table, and that requesting a second (different) model, by invoking the SAT solver with the formula $\varphi_2 = \varphi_1 \wedge \neg\theta_1$, results in the model θ_2. Notice that the variable x_1 takes the same value, *true*, in both models, and that the variable x_3 takes the same value, *false*. In the third iteration, the call $\varphi_3 = \varphi_2 \wedge (\neg x_1 \vee x_3)$ requests a model which is different from the first two and which flips the value of (at least) one of the two variables x_1 (to false) or x_3 (to true). Given that this call is not satisfiable, we conclude that x_1 and x_3 comprise the backbone variables of φ.

Now consider the case where in addition to the backbone we wish to derive also equations between literals which hold in all models of φ. The generalized backbones algorithm applied in BEE is basically the same algorithm as that proposed for computing backbones. Given formula φ, generalized backbones are computed by extending φ to φ' as prescribed by Equation (1). This is straightforward. Enumerating the variables of φ as $\{ x_1, \ldots, x_n \}$. One simply defines

$$\varphi' = \varphi \wedge \{ e_{ij} \leftrightarrow (x_i \leftrightarrow x_j) \mid 0 \leq i < j \leq n \} \tag{1}$$

introducing $\theta(n^2)$ fresh variables e_{ij}. If the literal e_{ij} is in the backbone of φ' then $x_i = x_j$ is implied by φ, and if the literal $\neg e_{ij}$ is in the backbone of φ' then $x_i = \neg x_j$ is implied by φ. As an optimization, it is possible to focus in the first two iterations only on the variables of φ. However, there is one major obstacle. The application of backbones with equalities for φ with n variables involves computing the backbone for φ' which has $\theta(n^2)$ variables. Consequently, it is reasonable to assume that the number of calls to the SAT solver may be quadratic. Below we prove that this is not the case, but first we present the following example.

	$\overbrace{x_1\ x_2\ x_3\ x_4\ x_5}$	$\overbrace{e_{12}\ e_{13}\ e_{14}\ e_{15}\ e_{23}\ e_{24}\ e_{25}\ e_{34}\ e_{35}\ e_{45}}$	φ_i
i=1	1 1 0 0 1	1 0 0 1 0 0 1 1 0 0	φ'
i=2	1 0 0 1 0	0 0 1 0 1 0 1 0 1 0	$\varphi_1 \wedge \neg\theta_1$
i=3	1 0 0 0 1	0 0 0 1 1 1 0 1 0 0	$\varphi_2 \wedge \begin{pmatrix} \neg x_1 \vee x_3 \vee e_{13} \vee \\ \neg e_{24} \vee \neg e_{25} \vee e_{45} \end{pmatrix}$
i=4	*unsat*		$\varphi_3 \wedge (\neg x_1 \vee x_3 \vee e_{13} \vee e_{45})$

Fig. 2. Demo of the generalized backbones algorithm (Example 2)

Example 2. Consider the same formula φ as in Example 1. Figure 2 illustrates the iterations, one per line, (four in this example) of the backbone algorithm but extended to operate on φ' as specified in Equation (1). The first two lines of the table in Figure 2 detail θ_1 and θ_2 which are almost the same as the two models of φ_1 and φ_2 from Figure 1, except that here we present also the values of the fresh variables e_{ij} indicating the values of the equations between literals. The two variables x_1 and x_3 (and hence also e_{13}) take identical values in both

models, just as in Figure 1. This time, in the third iteration we ask to either flip the value for one of $\{x_1, x_3\}$ or for one of $\{e_{13}, e_{24}, e_{25}, e_{45}\}$ and there is such a model θ_3 which flips the values of e_{24} and e_{25}. In the first three models the variables x_1, x_3, e_{13}, and e_{45} take single values. Hence in the fourth iteration the call to the SAT solver asks to flip one of them. Figure 2 indicates that this is not possible and hence these four variables are in the generalized backbone.

We proceed to prove that iterated SAT solving for generalized backbones using φ' involves at most $n+1$ satisfiable SAT tests, and exactly one unsatisfiable test, in spite of the fact that φ' involves a quadratic number of fresh variables.

Theorem 1. *Let φ be a CNF, X a set of n variables, and $\Theta = \{\theta_1, \ldots, \theta_m\}$ the sequence of assignments encountered by the generalized backbones algorithm for φ and X. Then, $m \leq n+1$.*

Before presenting a proof of Theorem 1 we introduce some terminology. Assume a set of Boolean variables X and a sequence $\Theta = \{\theta_1, \ldots, \theta_m\}$ of models. Denote $\hat{X} = X \cup \{1\}$ and let $x, y \in \hat{X}$. If $\theta(x) = \theta(y)$ for all $\theta \in \Theta$ or if $\theta(x) \neq \theta(y)$ for all $\theta \in \Theta$, then we say that Θ *determines* the equation $x = y$. Otherwise, we say that Θ *disqualifies* $x = y$, intuitively meaning that Θ disqualifies $x = y$ from being determined. More formally, Θ *determines* $x = y$ if and only if $\Theta \models (x = y)$ or $\Theta \models (x = \neg y)$, and otherwise Θ *disqualifies* $x = y$.

The generalized backbones algorithm for a formula φ and set of n variables X applies so that each iteration results in a satisfying assignment for φ which disqualifies at least one additional equation between elements of $\hat{X}$. Although there are a quadratic number of equations to be considered, we prove that the CEP algorithm terminates after at most $n+1$ iterations.

Proof. (of Theorem 1) For each value $i \leq m$, $\Theta_i = \{\theta_1, \ldots, \theta_i\}$ induces a partitioning, Π_i of $\hat{X}$ to disjoint and non-empty sets, defined such that for each $x, y \in \hat{X}$, x and y are in the same partition $P \in \Pi_i$ if and only if Θ_i determines the equation $x = y$. So, if $x, y \in P \in \Pi_i$ then the equation $x = y$ takes the same value in all assignments of Θ_i. The partitioning is well defined because if in all assignments of Θ_i both $x = y$ takes the same value and $y = z$ takes the same value, then clearly also $x = z$ takes the same value, implying that x, y, z are in the same partition of Π_i. Finally, note that each iteration $1 < i \leq m$ of the generalized backbones algorithm disqualifies at least one equation $x = y$ that was determined by Θ_{i-1}. This implies that at least one partition of Π_{i-1} is split into two smaller (non-empty) partitions of Π_i. As there are a total of $n+1$ elements in $\hat{X}$, there can be at most $n+1$ iterations to the algorithm.

Example 3. Consider the same formula φ as in Examples 1 and 2. Figure 3 illustrates the run of the algorithm in terms of the partitionings Π_i from the proof of Theorem 1 (in the right column of the table). There are four iterations, just as in Figure 2, and $\varphi_1, \ldots, \varphi_4$ and $\theta_1, \ldots, \theta_4$ are the same as in Figure 2, except that we do not make explicit all of the information in the table of Figure 3. To

	$\overbrace{x_1\ x_2\ x_3\ x_4\ x_5}^{\theta_i}$	Π_i
i=1	1 1 0 0 1	$\{ x_1, x_2, x_3, x_4, x_5, 1 \}$
i=2	1 0 0 1 0	$\{ x_1, x_3, 1 \}, \{ x_2, x_4, x_5 \}$
i=3	1 0 0 0 1	$\{ x_1, x_3, 1 \}, \{ x_2 \}, \{ x_4, x_5 \}$
i=4	*unsat*	

Fig. 3. Demo of the generalized backbone algorithm performing a linear number of calls as in the proof of Theorem 1 (Example 3)

better understand the meaning of the partition observe for instance the second iteration where Π_2 indicates that the set $\Theta_2 = \{\theta_1, \theta_2\}$: (a) determines the equations between $\{x_1, x_3, 1\}$ (for both assignments, $x_1 = 1$ is *true*, $x_3 = 1$ is *false*, and $x_1 = x_3$ is false), and (b) determines the equations between $\{x_2, x_4, x_5\}$ (for both assignments, $x_2 = x_4$, $x_4 = x_5$, and $x_5 = x_2$ are *false*).

4 Complete Equi-propagation in BEE

BEE is a compiler which facilitates solving finite domain constraints by encoding them to CNF and applying an underlying SAT solver. In BEE constraints are modeled as Boolean functions which propagate information about equalities between Boolean literals. This information is then applied to simplify the CNF encoding of the constraints. This process is termed *equi-propagation*. A key factor is that considering only a small fragment of a constraint model at one time enables to apply stronger, and as we argue in this paper, even complete reasoning to detect equivalent literals in that fragment. Once detected, equivalences propagate to simplify the entire constraint model and facilitate further reasoning on other fragments. BEE is described in several recent papers: [14], [13] and [15].

In BEE, each constraint is associated with a collection of simplification rules. The simplification of one constraint, may result in that we derive an equality of the form $x = \ell$, where x is a Boolean variable and ℓ a Boolean literal or constant, which is implied by one or more of the given constraints. The compiler then propagates this equality to other constraints, which may in turn trigger simplification rules for additional constraints. BEE iterates until no further rules apply.

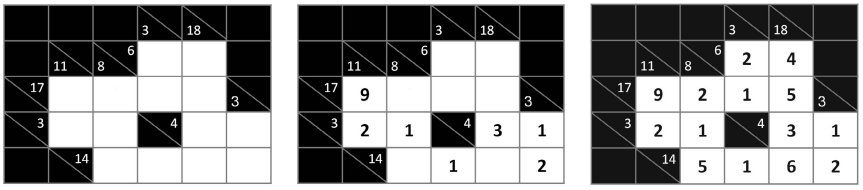
Figure 4 illustrates a constraint model in BEE for the Kakuro instance depicted as Figure 5(a). In a Kakuro instance each block of consecutive horizontal or vertical white cells must be filled with distinct non-zero digits (integer values between 1 and 9) which sum to a given clue. For example, in the bottom row of Figure 5(a) the four cells must sum to 14 and a possible solution is to assign them the distinct values 5,1,6,2. The constraints in the left column of Figure 4 declare the finite domain variables of the instance. Each cell is associated with a finite domain variable taking values between 1 and 9. Each block of integer variables (horizontal and vertical) in the instance is associated with a `int_array_plus` constraint (middle column) and an `allDiff` constraint (right column).

The simplification rules defined in BEE consider each individual constraint to determine (generalized) backbone variables in its underlying bit-level representation. These rules are "ad-hoc" in that they do not derive all of the equalities implied by a constraint. Moreover if an equation is implied by a set of constraints but not by an individual constraint, then BEE may not detect that equation. Figure 5(b) illustrates the effect of applying BEE to the constraint model of Figure 4 and demonstrates that 7 of the 14 integer variables in the instance are determined at compile time.

In this paper we propose to enhance BEE to allow the user to specify sets of constraints to which a generalized backbone algorithm is to be applied. We call this process complete equi-propagation (CEP). Figure 5(c) illustrates the effect of the enhanced BEE where each pair of constraints about a given block: one `int_array_plus` and one `allDiff` are grouped for CEP. Here we see that all 14 integer variables in the instance are determined at compile time.

$\texttt{new\_int}(I_1, 1, 9)$	$\texttt{int\_array\_plus}([I_1, I_2], 6)$	$\texttt{allDiff}([I_1, I_2])$
$\texttt{new\_int}(I_2, 1, 9)$	$\texttt{int\_array\_plus}([I_3, I_4, I_5, I_6], 17)$	$\texttt{allDiff}([I_3, I_4, I_5, I_6])$
$\texttt{new\_int}(I_3, 1, 9)$	$\texttt{int\_array\_plus}([I_7, I_8], 3)$	$\texttt{allDiff}([I_7, I_8])$
$\texttt{new\_int}(I_4, 1, 9)$	$\texttt{int\_array\_plus}([I_9, I_{10}], 4)$	$\texttt{allDiff}([I_9, I_{10}])$
$\texttt{new\_int}(I_5, 1, 9)$	$\texttt{int\_array\_plus}([I_{11}, I_{12}, I_{13}, I_{14}], 14)$	$\texttt{allDiff}([I_{11}, I_{12}, I_{13}, I_{14}])$
$\texttt{new\_int}(I_6, 1, 9)$	$\texttt{int\_array\_plus}([I_3, I_7], 11)$	$\texttt{allDiff}([I_3, I_7])$
$\texttt{new\_int}(I_7, 1, 9)$	$\texttt{int\_array\_plus}([I_4, I_8, I_{11}], 8)$	$\texttt{allDiff}([I_4, I_8, I_{11}])$
$\texttt{new\_int}(I_8, 1, 9)$	$\texttt{int\_array\_plus}([I_1, I_5], 3)$	$\texttt{allDiff}([I_1, I_5])$
$\texttt{new\_int}(I_9, 1, 9)$	$\texttt{int\_array\_plus}([I_2, I_6, I_9, I_{13}], 18)$	$\texttt{allDiff}([I_2, I_6, I_9, I_{13}])$
$\texttt{new\_int}(I_{10}, 1, 9)$	$\texttt{int\_array\_plus}([I_{10}, I_{14}], 3)$	$\texttt{allDiff}([I_{10}, I_{14}])$
$\texttt{new\_int}(I_{11}, 1, 9)$		
$\texttt{new\_int}(I_{12}, 1, 9)$		
$\texttt{new\_int}(I_{13}, 1, 9)$		
$\texttt{new\_int}(I_{14}, 1, 9)$		

Fig. 4. Constraints for the Kakuro instance of Figure 5(a)



a. Kakuro Instance b. Specialized by BEE c. Specialized with CEP

Fig. 5. Applying complete equi-propagation to a Kakuro Instance

The compilation of a constraint model to a CNF using BEE goes through three phases. In the first phase, bit blasting, integer variables (and constants) are represented as bit vectors. Now all constraints are about Boolean variables. The

second phase, the main loop of the compiler, is about constraint simplification. Three types of actions are applied: ad-hoc equi-propagation, partial evaluation, and decomposition of constraints. Simplification is applied repeatedly until no rule is applicable. In the third, and final phase, simplified constraints are encoded to CNF.

In order to enhance BEE with CEP we introduce syntax for users to specify constraints or groups of constraints designated for CEP. We introduce a new phase to the compilation process which is applied after the first phase (bit blasting) and before the second phase. In the new phase, we first apply two types of actions, ad-hoc equi-propagation and partial evaluation, repeatedly until no rule is applicable. Then we apply generalized backbone computation on the CNF of the user defined groups of constraints. If new equalities are derived from applying the CEP algorithm on one of the user defined groups, then these are propagated to all relevant constraints and the simplification process is repeated until no more equalities are derived.

At the syntactic level, we have added two constructs to BEE: `cep(C)` specifies that the CEP algorithm is applied to the CNF of constraint `C` instead of the ad-hoc rules in BEE; and `cep_group(G)` specifies that CEP is applied to the conjunction of CNF's corresponding to the constraints in a group `G`. For instance, the user might specify `cep_group([int_array_plus([I1, I2], 6), allDiff([I1, I2])])` for pairs of constraints in the Kakuro example.

5 Preliminary Experimental Evaluation

We demonstrate the potential of CEP with 2 experiments. Our goal is to illustrate the impact of CEP on the BEE constraint compiler. We demonstrate that, in some cases, enhancing the compilation of constraints to consider CEP (using a SAT solver) at compile time leads to much better results than those obtained when using BEE's polynomial time compilation based on ad-hoc equi-propagation rules. To this end we view CEP as yet one more compilation tool that the user can choose to apply.

5.1 The First Experiment

We illustrate the impact of CEP with an application of extremal graph theory where the goal is to find the largest number of edges in a simple graph with n nodes such that any cycle (length) is larger than 4. The graph is represented as a Boolean adjacency matrix A and there are three types of constraints:

1. Constraints about cycles in the graph: $\forall_{i,j,k}.\ A[i,j] + A[j,k] + A[k,i] < 3$, and $\forall_{i,j,k,l}.\ A[i,j] + A[j,k] + A[k,l] + A[l,i] < 4$;
2. Constraints about symmetries: in addition to the obvious $\forall_{1 \leq i < j \leq n}.\ (A[i,j] \equiv A[j,i] \quad \text{and} \quad A[i,i] \equiv \mathit{false})$, we constrain the rows of the adjacency matrix to be sorted lexicographically (justified in [1]); and
3. Constraints that impose lower and upper bounds on the degrees of the graph nodes as described in [3].

Further details on this benchmark can be found in [1].

Table 1 illustrates results, running BEE with and without CEP. Here, we focus on finding a graph with the prescribed number of graph nodes with the known maximal number of edges (all instances are satisfiable). In our encoding, if CEP is applied, then there is a single CEP group consisting of all of the constraints related to symmetry breaking and node degrees (items 2 and 3 detailed above). All of the other constraints related to graph cycles (item 1 detailed above) remain ungrouped. The table details for each instance:

- in the first two columns: the number of nodes and edges;
- in the next two columns: the size of the single CEP group which is input to the CEP algorithm (number of clauses and variables, after ad-hoc simplification by BEE);
- in the next two sets of 4 columns: for each CEP choice: the BEE compilation time, the (total) number of clauses and variables in the target CNF encoding, and the subsequent sat solving time.

The table indicates that CEP increases the compilation time (within reason), reduces the CNF size (considerably), and (for the most part) improves SAT solving time.[2] The CEP groups in this example involve, in average, about half the number of variables and one third the number of clauses when compared to the encoding without CEP, with up to 10,000 variables and 50,000 clauses. We note, that as explained above, BEE iterates over CEP groups, propagating equations learned from one group to simplify other groups which as a result may yield further CEP information. The group sizes presented in Table 1 indicate the initial group sizes. During iteration the size of the groups decreases.

5.2 The Second Experiment

We consider the 12 instances from the `armies` benchmark from category DEC-SMALLINT-LIN of the PB'12 competition.[3] Each instance consists of a set of cardinality and pseudo Boolean constraints.

For this experiment we consider a grouping of the constraints into two (overlapping) groups. These are obtained by considering a connectivity graph where the nodes are constraint variables. There is a weighted edge (u, v) with weight $dep(u, v)$ between variables u and v if their degree of dependency $dep(u, v) > 0$ which is defined as follows:

Given a set of constraints Cs. Let V be the set of (Boolean and integer) variables occurring in Cs. For a variable $v \in V$, let $C(v)$ denote the set of constraints that involve v. For a pair of variables $u, v \in V$, $dep(u, v) = |C(u) \cap C(v)|$ is a simple measure on the potential dependency between u and v.

We also define $level(v) = max \left\{ dep(u, v) \mid u \in V \right\}$, a measure on the dependencies of variable v: a variable v has a high dependency level if there is another

[2] Experiments are performed on a single core of an Intel(R) Core(TM) i5-2400 3.10GHz CPU with 4GB memory under Linux (Ubuntu lucid, kernel 2.6.32-24-generic).

[3] See `http://www.cril.univ-artois.fr/PB12/`

Table 1. Search for graphs with no cycles of size 4 or less (comp. & solve times in sec.)

		group size		with CEP				without CEP			
nodes	edges	clauses	vars	comp.	clauses	vars	solve	comp.	clauses	vars	solve
15	26	8157	1761	0.24	13421	2154	0.07	0.10	23424	3321	0.08
16	28	10449	2200	0.26	18339	2851	0.19	0.12	30136	4328	0.34
17	31	11996	2558	0.39	21495	3233	0.07	0.16	37074	5125	0.12
18	34	14235	3030	0.49	26765	3928	0.12	0.21	45498	6070	0.13
19	38	16272	3433	0.46	30626	4380	0.11	0.22	54918	7024	0.15
20	41	21307	4354	0.55	43336	6005	5.93	0.25	68225	8507	12.70
21	44	24715	5037	0.77	52187	7039	1.46	0.31	81388	9835	69.46
22	47	28296	5762	0.88	61611	8118	71.73	0.35	96214	11276	45.43
23	50	32213	6556	1.10	73147	9352	35.35	0.38	113180	13101	27.54
24	54	35280	7278	2.02	81634	10169	96.11	0.50	130954	14712	282.99
25	57	40942	8344	1.40	99027	12109	438.91	0.53	152805	16706	79.11
26	61	44817	9208	4.58	110240	13143	217.72	0.73	175359	18615	815.55
27	65	50078	10282	2.16	127230	14856	35.36	0.75	201228	20791	114.55

variable u such that u and v occur together in a large number of constraints. For a grouping, we first take all of the constraints which contain a variable of maximal level. Denote $max = max \left\{ \; level(v) \;\middle|\; v \in V \; \right\}$ and define

$$G_1 = \left\{ \; c \;\middle|\; v \in V,\; level(v) = max,\; c \in C(v) \; \right\}$$
$$G_2 = Cs \setminus G_1$$

In the context of our experiments G_1 was small in comparison to G_2, and we extended it by defining

$$G_1' = \left\{ \; c \;\middle|\; v \in V,\; level(v) \geq k,\; c \in C(v) \; \right\}$$

for a suitable value $k < max$ such that the size of G_1' and G_2 are more or less equal. Note that we did not redefine G_2 so the two groups overlap. We found this redundancy useful.

Table 2 illustrates the encoding sizes for the instances of the `armies` benchmark. In the first 2 columns we indicate the instance name and the number of original instance variables (ι-vars). Next we illustrate the encoding sizes for three configurations: using BEE (with ad-hoc equi-propagation), using BEE with a backbone algorithm, and using BEE with a CEP algorithm. For backbones and CEP we group constraints into two groups (G_1' and G_2 described above), more or less equal in size. For encoding sizes we indicate the number of CNF variables and clauses as well as the percent of the original instance variables eliminated during the compilation process ($\Delta\iota$). So in average, BEE with CEP eliminates 6.3% of the original instance variables, while BEE with backbones eliminates 3.7% and BEE alone (with its ad-hoc techniques) eliminates 2.0%. These reductions, although small, may be significant as we are eliminating variables on the input to the SAT solver.

Table 2. The `Army` benchmarks — Encoding sizes

instance		BEE (regular)			BEE (backbones)			BEE (CEP)		
name	ι-vars	*vars*	*clauses*	$\Delta\iota$	*vars*	*clauses*	$\Delta\iota$	*vars*	*clauses*	$\Delta\iota$
$8 \times 9ls$	220	2136	8072	0.0	2136	8072	0.0	2096	7932	3.6
$8 \times 9bt$	220	2893	11567	5.0	2488	9962	10.5	2438	9784	15.0
$9 \times 12ls$	266	2778	11077	0.0	2778	11077	0.0	2738	10937	3.0
$9 \times 12bt$	266	3522	14441	4.5	3204	13101	7.1	3172	12994	10.2
$10 \times 14ls$	316	4008	14450	0.0	4008	14450	0.0	3978	14345	1.9
$11 \times 17ls$	370	4914	18447	0.0	4914	18447	0.0	4874	18307	2.2
$10 \times 14bt$	316	6433	25553	4.1	5711	22969	8.5	5657	22771	11.7
$11 \times 17bt$	370	7446	29425	3.8	6914	27290	5.9	6882	27183	8.1
$13 \times 24ls$	490	7240	28645	0.0	7240	28645	0.0	7210	28541	1.2
$13 \times 24bt$	490	15298	60988	3.3	13858	55675	5.1	13826	55568	6.7
$12 \times 21bt$	428	14094	55614	3.5	12515	49725	7.2	12457	49507	9.6
$12 \times 21ls$	428	5912	23144	0.0	5912	23144	0.0	5872	23004	1.9
Average		6390	25119	2.0	5973	23546	3.7	5933	23406	6.3

Table 3. The `Army` benchmarks — solving times (seconds) with 1800 sec. timeout

instance	BEE (regular)			BEE(Backbones)			BEE (CEP)		
name	*comp.*	*solve*	*total*	*comp.*	*solve*	*total*	*comp.*	*solve*	*total*
$8 \times 9ls$	0.0	0.6	0.6	0.4	0.1	0.5	0.7	0.5	1.2
$8 \times 9bt$	0.1	1.0	1.1	0.7	0.6	1.3	1.3	0.7	2.0
$9 \times 12ls$	0.1	5.9	6.0	0.5	3.0	3.5	1.0	1.1	2.1
$9 \times 12bt$	0.1	3.4	3.5	1.0	7.5	8.5	1.4	1.7	3.1
$10 \times 14ls$	0.1	17.0	17.1	0.7	1.0	1.7	1.5	8.2	9.7
$11 \times 17ls$	0.1	4.5	4.6	0.9	65.4	66.3	1.8	8.1	9.9
$10 \times 14bt$	0.1	45.2	45.3	1.9	100.4	102.3	4.1	6.9	11.0
$11 \times 17bt$	0.2	267.8	268.0	2.5	344.0	346.5	4.0	26.9	30.9
$13 \times 24ls$	0.1	833.8	833.9	2.2	880.8	883.0	3.5	652.2	655.7
$13 \times 24bt$	0.3	1072.7	1073.0	8.9	∞	∞	14.9	768.7	783.6
$12 \times 21bt$	0.3	∞	∞	7.0	∞	∞	18.9	∞	∞
$12 \times 21ls$	0.1	∞	∞	1.7	∞	∞	3.1	∞	∞

Table 3 details the solving times for the three encodings of the army benchmarks: using BEE (with ad-hoc equi-propagation), using BEE enhanced with backbones, and using BEE enhanced with generalized backbones (CEP). For each encoding and instance we indicate the compile time, the SAT solving time, and the total solving time (sum of the previous two).

First, we comment on compile times. Compile times increase for backbones and even more for generalized backbones. But this increase comes with a discount in solving time. Consider the instance $13 \times 24bt$. Compilation time for CEP goes up from 0.3 sec. to 14.9 sec, but total solving time goes down from 1073 sec. to 784 sec.

These instances are hard, although they are relatively small. The largest instance, $13 \times 24bt$, has only 490 (Boolean) variables and involves 1085 constraints.

Only four out of the 42 solvers tested on these instances in the PB'12 competition manage to solve instance $13 \times 24bt$ within the 1800 second timeout (the same timeout is used in our experiments). From these four solvers, the two SAT-based solvers require more or less 1600 seconds (on the competition machines) whereas our solution for this instance is under 800 seconds.

6 Conclusion

This paper generalizes the notion of a CNF backbone to capture also equalities between literals. We prove that computing generalized backbones, just as usual backbones, involves only a linear number of calls to a SAT solver. We describe the integration of backbones with equality to enhance the BEE constraint-to-CNF compiler and demonstrate the utility of our approach through a preliminary experimentation.

Acknowledgments: We thank Peter Stuckey for helpful discussions and comments.

References

1. Codish, M., Miller, A., Prosser, P., Stuckey, P.J.: Breaking symmetries in graph representation. In: Rossi, F. (ed.) IJCAI. IJCAI/AAAI (2013)
2. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
3. Garnick, D.K., Kwong, Y.H.H., Lazebnik, F.: Extremal graphs without three-cycles or four-cycles. Journal of Graph Theory 17(5), 633–645 (1993)
4. Heule, M.J.H., Järvisalo, M., Biere, A.: Efficient CNF simplification based on binary implication graphs. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 201–215. Springer, Heidelberg (2011)
5. Heule, M., van Maaren, H.: Aligning CNF- and equivalence-reasoning. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 145–156. Springer, Heidelberg (2005)
6. Janota, M.: SAT Solving in Interactive Configuration. PhD thesis, University College Dublin (November 2010)
7. Kilby, P., Slaney, J.K., Thiébaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: Veloso, M.M., Kambhampati, S. (eds.) AAAI, pp. 1368–1373. AAAI Press / The MIT Press (2005)
8. Kirkpatrick, S., Toulouse, G.: Configuration space analysis of traveling salesman problems. J. Phys. (France) 46, 1277–1292 (1985)
9. Li, C.-M.: Equivalent literal propagation in the DLL procedure. Discrete Applied Mathematics 130(2), 251–276 (2003)
10. Manolios, P., Papavasileiou, V.: Pseudo-boolean solving by incremental translation to SAT. In: Bjesse, P., Slobodová, A. (eds.) FMCAD, pp. 41–45. FMCAD Inc. (2011)
11. Manthey, N.: Coprocessor 2.0 - a flexible CNF simplifier - (tool presentation). In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 436–441. Springer, Heidelberg (2012)

12. Marques-Silva, J., Janota, M., Lynce, I.: On computing backbones of propositional theories. In: Coelho, H., Studer, R., Wooldridge, M. (eds.) ECAI. Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 15–20. IOS Press (2010), Extended version: `http://sat.inesc-id.pt/~mikolas/bb-aicom-preprint.pdf`
13. Metodi, A., Codish, M.: Compiling finite domain constraints to SAT with BEE. TPLP 12(4-5), 465–483 (2012)
14. Metodi, A., Codish, M., Lagoon, V., Stuckey, P.J.: Boolean equi-propagation for optimized SAT encoding. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 621–636. Springer, Heidelberg (2011)
15. Metodi, A., Codish, M., Stuckey, P.J.: Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. J. Artif. Intell. Res. (JAIR) 46, 303–341 (2013)
16. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: Determining computational complexity for characteristic phase transitions. Nature 400, 133–137 (1998)
17. Schneider, J.J.: Searching for backbones – an efficient parallel algorithm for the traveling salesman problem. Comput. Phys. Commun. (1996)
18. Sheeran, M., Stålmarck, G.: A tutorial on stålmarck's proof procedure for propositional logic. Formal Methods in System Design 16(1), 23–58 (2000)
19. Stålmark, G.: A system for determining propositional logic theorem by applying values and rules to triplets that are generated from a formula. US Patent 5,276,897; Canadian Patent 2,018,828; European Patent 0403 545; Swedish Patent 467 076 (1994)
20. Zhang, W.: Phase transitions and backbones of the asymmetric traveling salesman problem. J. Artif. Intell. Res. (JAIR) 21, 471–497 (2004)