Towards Robust CNF Encodings of Cardinality Constraints

Joao Marques-Silva[1] and Inês Lynce[2]

[1] School of Electronics and Computer Science, University of Southampton, UK
jpms@ecs.soton.ac.uk
[2] IST/INESC-ID, Technical University of Lisbon, Portugal
ines@sat.inesc-id.pt

Abstract. Motivated by the performance improvements made to SAT solvers in recent years, a number of different encodings of constraints into SAT have been proposed. Concrete examples are the different SAT encodings for $\leq 1\ (x_1, \ldots, x_n)$ constraints. The most widely used encoding is known as the pairwise encoding, which is quadratic in the number of variables in the constraint. Alternative encodings are in general linear, and require using additional auxiliary variables. In most settings, the pairwise encoding performs acceptably well, but can require unacceptably large Boolean formulas. In contrast, linear encodings yield much smaller Boolean formulas, but in practice SAT solvers often perform unpredictably. This lack of predictability is mostly due to the large number of auxiliary variables that need to be added to the resulting Boolean formula. This paper studies one specific encoding for $\leq 1\ (x_1, \ldots, x_n)$ constraints, and shows how a state-of-the-art SAT solver can be adapted to overcome the problem of adding additional auxiliary variables. Moreover, the paper shows that a SAT solver may essentially ignore the existence of auxiliary variables. Experimental results indicate that the modified SAT solver becomes significantly more robust on SAT encodings involving $\leq 1\ (x_1, \ldots, x_n)$ constraints.

1 Introduction

In recent years SAT solvers have increasingly been used in practical applications, including planning, hardware and software model checking, among many others. The encoding of an increasing number of computational problems into SAT raises the challenge of encoding different constraints into SAT. Among the constraints for which dedicated CNF encodings have been proposed, a special emphasis has been given to cardinality constraints [2, 3]. In addition, past work addressed special forms of cardinality constraints, including $\leq 1\ (x_1, \ldots, x_n)$ constraints [11, 13, 1]. Observe that, given the straightforward CNF encoding of $\geq 1\ (x_1, \ldots, x_n)$ constraints, the encoding of $\leq 1\ (x_1, \ldots, x_n)$ constraints also serves for encoding $= 1\ (x_1, \ldots, x_n)$ constraints. In practice, the constraints $\geq 1\ (x_1, \ldots, x_n)$, $\leq 1\ (x_1, \ldots, x_n)$ and $= 1\ (x_1, \ldots, x_n)$ find a large number of applications.

A number of alternative encodings have been proposed for $\leq 1\ (x_1, \ldots, x_n)$ constraints. Encodings can be linear, logarithmic or quadratic, in the number

of variables, and may or may not guarantee arc-consistency. The most widely used encoding, that is often referred to as the pairwise encoding, is quadratic in the number of variables in the constraint and guarantees arc-consistency. Interestingly, most available benchmarks containing $\leq 1\ (x_1, \ldots, x_n)$ constraints use the pairwise encoding.

In most settings, the pairwise encoding performs acceptably well, but can require unacceptably large Boolean formulas. In contrast, linear encodings yield much smaller Boolean formulas, but in practice SAT solvers often perform unpredictably. This lack of predictability is mostly due to the large number of auxiliary variables that need to be added to the resulting Boolean formula.

This paper addresses one concrete linear encoding for $\leq 1\ (x_1, \ldots, x_n)$ constraints which guarantees arc-consistency [21], and identifies several properties associated with the auxiliary variables used. One consequence is that a SAT solver may essentially ignore the existence of auxiliary variables, and so this indirectly overcomes the problem of adding additional auxiliary variables. Experimental results indicate that a SAT solver that filters auxiliary variables becomes significantly more robust.

The paper is organized as follows. The next section surveys encodings for $\leq 1\ (x_1, \ldots, x_n)$ constraints. Section 3 provides a brief perspective of recent backtracking SAT solvers, referred to as conflict-driven clause-learning (CDCL) SAT solvers. Section 4 outlines some of the properties of the sequential counter encoding of [21]. Experimental results are analyzed in section 5. The paper concludes in section 6, by analyzing how some of the results proposed in the paper can be extended to other cardinality constraints and encodings, and by outlining directions for future research.

2 Related Work

A large body of research exists on encoding constraints into CNF [16, 23, 22, 11, 13, 12, 6, 1, 10, 4]. In addition, dedicated encodings have been proposed for specific types of constraints, including cardinality constraints [2, 3].

A special case of cardinality constraints are those of the form $\leq 1\ (x_1, \ldots, x_n)$, which are widely used in practice. The most often used CNF encoding for $\leq 1\ (x_1, \ldots, x_n)$ constraints is referred to as the pairwise encoding. Given a $\leq 1\ (x_1, \ldots, x_n)$ constraint, the pairwise encoding is formulated as follows:

$$\bigwedge_{\substack{S \subseteq \{1, \ldots, n\} \\ |S| = 2}} \left(\bigvee_{j \in S} \neg x_j \right) \tag{1}$$

This encoding introduces no additional auxiliary variables, but grows quadratically with the number of variables in the constraint.

An alternative is to use additional variables, thus obtaining asymptotically more efficient encodings. A number of linear encodings has been proposed over

the years [11, 13, 12, 1, 21, 23]. Some of these encodings do not guarantee arc-consistency (e.g. [23]) whereas others do [11, 21].

One recent CNF encoding for cardinality constraints $\leq k$ $(x_1, \ldots, x_n)$ is based on sequential counters [21]. The resulting CNF encoding is denoted by $LT^{n,k}_{SEQ}$. A special case of cardinality constraints is considered, namely ≤ 1 $(x_1, \ldots, x_n)$ constraints. The associated encoding will be denoted by $LT^{n,1}_{SEQ}$.

An arbitrary number of ≤ 1 $(x_1, \ldots, x_n)$ constraints is assumed, each being represented by an index k. Hence, each constraint is of the form:

$$\sum_{i=1}^{n_k} x_i^k \leq 1 \tag{2}$$

Where n_k is the number of variables in the constraint. From [21], the CNF encoding for the above constraint becomes:

$$(\neg x_1^k \vee s_1^k) \wedge (\neg x_n^k \vee \neg s_{n-1}^k) \bigwedge_{1<i<n_k} \big((\neg x_i^k \vee s_i^k) \wedge (\neg s_{i-1}^k \vee s_i^k) \wedge (\neg x_i^k \vee \neg s_{i-1}^k)\big) \tag{3}$$

Where s_i^k, $1 \leq k \leq n-1$, are auxiliary variables. When clear from context, the index k is dropped, and so the $LT^{n,1}_{SEQ}$ encoding becomes:

$$(\neg x_1 \vee s_1) \wedge (\neg x_n \vee \neg s_{n-1}) \bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1})) \tag{4}$$

The remainder of the paper focus on the sequential counter CNF encoding and shows that this encoding has a number of interesting properties that can be exploited by a clause learning SAT solver. Before, however, the organization of backtracking SAT solvers is briefly overviewed.

3 CDCL SAT Solvers

This section provides a necessarily brief perspective of modern CDCL SAT solvers. CDCL SAT solvers follow the organization of the DPLL algorithm [8, 7], but integrate a number of effective techniques, including clause learning [17, 5], lazy data structures [18] and search restarts [15]. CDCL SAT solvers have evolved from the original solvers [17, 5, 24], which essentially proposed clause learning, to the more recent CDCL SAT solvers, that also integrate lazy data structures and search restarts [18, 14, 9].

In the following sections, a number of concepts associated with CDCL SAT solvers will be used. These concepts are briefly reviewed below (see [17, 18, 14, 9] for additional detail).

A CDCL SAT solver is usually organized into three main engines [17, 18, 9]: the decision engine, used for branching; the deduction engine, used for unit propagation and identification of unsatisfied clauses (or *conflicts*); and the diagnosis engine, used for clause learning.

A *decision level* is associated with each assigned variable. Decision levels measure the depth of the search tree in terms of the number of variables the SAT algorithm has branched on. Variables can be assigned a Boolean value, either resulting from a decision (or branching step), or as the result of unit propagation [8]. Variables assigned as the result of unit propagation are said to be *implied*. With each implied variable the SAT algorithm also associates a *reason* or *antecedent*, representing the clause that explains why the variable is implied. The set of assigned variables and associated reasons implicitly represent the *implication graph* [17].

The process of *clause learning* consists of traversing the implication graph from a given unsatisfied clause by using the reasons of implied variable assignments, and recording unsatisfied literals assigned at decision levels less than the current one. The resulting set of recorded literals is then used to create a new clause, which serves for backtracking non-chronologically, and for preventing the same conflict from occurring again during the search process.

Moreover, all effective CDCL solvers use unique implication points (UIPs) [17, 25]. UIPs represent dominators in the implication graph of unsatisfied clauses with respect to the most recent decision variable. Whereas some of the early CDCL SAT solvers would use UIPs to learn more clauses [17], more recent CDCL SAT solvers stop clause learning at the first UIP [25]. Albeit stopping at the first UIP usually yields a larger number of decision steps, it is also an observed fact that a smaller number of learnt clauses results in faster execution, and most often this results in smaller run times [25].

Finally, and besides the hallmarks of all CDCL SAT solvers, a number of additional techniques have been quite successfully used in recent solvers. These include deletion policies for learnt clauses [14, 9, 20], techniques for organization of literals in learnt clauses [19], and the representation of binary clauses as direct implications [20].

4 Filtering Auxiliary Variables

This section revisits the $LT^{n,1}_{SEQ}$ CNF encoding (see section 2), and introduces a number of its properties. Some of these properties allow eliminating most (or even all) of the auxiliary variables for branching purposes. Moreover, this section also outlines how to adapt a SAT solver for filtering auxiliary variables.

4.1 Analysis of the $\leq 1\ (x_1, \ldots, x_n)$ Encoding

This section identifies a number of properties of the $LT^{n,1}_{SEQ}$ encoding. These properties essentially allow a SAT solver to ignore all (or at least most) of the auxiliary variables.

The first property is used throughout this section, for proving additional properties of the $LT^{n,1}_{SEQ}$ encoding.

Proposition 1. *Consider the CNF encoding $LT^{n,1}_{SEQ}$ of constraint $\leq 1\,(x_1, \ldots, x_n)$. Furthermore, assume that all x and s variables are unassigned, and that a given variable x_i is assigned value 1. Then, the following holds:*

1. *All s_j variables, with $1 \leq j < i$, are implied to value 0.*
2. *All s_j variables, with $i \leq j \leq n-1$, are implied to value 1.*
3. *All x_j variables, with $j \neq i$, are implied to value 0;*

Proof. The result follows from the analysis of (4). The third clause guarantees that s_i is implied value 1. Subsequently, the fourth clause implies that every s_j, with $n-1 \geq j > i$, is implied value 1. The fifth clause guarantees that s_{i-1} is implied value 0. Subsequently, the fourth clause implies that every s_j, with $i > j \geq 1$, is implied value 0. Analysis of the first and last case is also straightforward. Finally, the third and fourth clauses ensure that every x_j, with $j \neq i$, is implied value 0. Again, analysis of the first and last case is straightforward. □

The next step is to evaluate the role of the auxiliary variables used in the $LT^{n,1}_{SEQ}$ encoding.

Proposition 2. *For any complete satisfying assignment to the variables $x_1, \ldots, x_n$ of (2), the following holds:*

1. *All clauses of (4) containing literals of x variables are satisfied.*
2. *There exist assignments to the auxiliary variables $s_1, \ldots s_n$ that satisfy the clauses of (4) containing no literals of x variables.*

Proof. If (2) is satisfied, then at most one of the x_i variables is assigned value 1. Hence, two cases need to be considered: either all x_i variables are assigned value 0, or exactly one variable x_i is assigned value 1 and the remaining x variables are assigned value 0. Now consider the $LT^{n,1}_{SEQ}$ (4) encoding of (2). For the first case, the x_i variables satisfy all clauses that contain a literal in an x variable. Hence, only the clauses $(\neg s_{i-1} \vee s_i)$ need to be satisfied, and this can be achieved by assigning value 0 to all s_i variables. For the second case, exactly one variable x_i is assigned value 1. (4) ensures that all auxiliary variables are assigned a given value. This is immediate from proposition 1. □

The previous result guarantees that by branching only of non-auxiliary variables, either a satisfying assignment exists, in which case it is simple to find consistent assignments to the auxiliary variables, or no satisfying assignment exists, in which case it is unnecessary to branch on the auxiliary variables.

It is possible to extend the previous result further, by analyzing the clause learning process of a SAT solver, when branching is restricted to the non-auxiliary variables. As a result, in what follows, the SAT solver is assumed to branch *only* on non-auxiliary variables.

The following results assert that, when branching is restricted *only* to non-auxiliary variables, the participation of auxiliary variables in conflicts is fairly constrained. This allows effectively discarding auxiliary variables from learnt clauses.

Proposition 3. *For the CNF encoding $LT^{n,1}_{SEQ}$ of constraint $\leq 1\ (x_1, \ldots, x_n)$, if the value of an auxiliary variable s_i is implied at decision level l, then all other auxiliary variables associated with the same constraint are implied at the same decision level, or a conflict is identified.*

Proof. From (4), the value of an auxiliary variable s is implied by the value of an x variable only when the x variable is assigned value 1. Without loss of generality, let x_i be the variable that is assigned value 1. Then, all s variables with index no less than i are assigned value 1, and all s variables with index less than i are assigned value 0 (see proposition 1). If more than one x variable is assigned value 1, then a conflict is identified. □

Proposition 4. *For the CNF encoding $LT^{n,1}_{SEQ}$ of constraint $\leq 1\ (x_1, \ldots, x_n)$, auxiliary variables can only be implied by single implication paths from non-auxiliary variables.*

Proof. For any CNF encoding $LT^{n,1}_{SEQ}$ of constraint $1\ (x_1, \ldots, x_n)$, all auxiliary variables are assigned by unit propagation on binary clauses. Hence, the result follows. □

Proposition 5. *For the CNF encoding $LT^{n,1}_{SEQ}$ of constraint $\leq 1\ (x_1, \ldots, x_n)$, learnt clauses can contain a single auxiliary variable s_i, which is a UIP variable.*

Proof. (Sketch)[3] The clause learning algorithm used by most SAT solvers [17, 18, 14, 9] records literals from clauses traced during the conflict analysis procedure. These literals must be assigned at decision levels less than the current decision level, or otherwise the literal corresponds to the UIP variable.
Since all auxiliary variables are assigned by unit propagation on binary clauses, and all must be assigned at the most recent decision level, then the conflict analysis procedure cannot record literals associated with auxiliary variables, unless variable tracing stops at a UIP corresponding to an auxiliary variable. □

Proposition 6. *For the CNF encoding $LT^{n,1}_{SEQ}$ of constraint $\leq 1\ (x_1, \ldots, x_n)$, and during clause learning, if a s_i variable is a UIP, then it can be replaced by a single non-auxiliary variable x_k.*

Proof. (Sketch) Proposition 4 guarantees that auxiliary variables are implied as the result of a single implication path. Hence, it suffices to trace this single implication path to eventually reach a single non-auxiliary variable. □

Proposition 7. *For the CNF encoding $LT^{n,1}_{SEQ}$ of constraint $\leq 1\ (x_1, \ldots, x_n)$, any learnt clause ω can be replaced by a learnt clause of the same size that only contains non-auxiliary variables.*

Proof. From proposition 5, auxiliary variables can occur in learnt clauses only if they are traced as a UIP. From proposition 6 this traced variable can be replaced by a single non-auxiliary variable. Hence the result follows. □

[3] A detailed proof would require a more formal definition of the organization of a CDCL SAT solver.

As a result, the main conclusion is that by branching only on non-auxiliary variables, then a satisfying assignment can be identified, if it exists. Otherwise, by branching only on non-auxiliary variables, unsatisfiability can be proved.

Moreover, by analyzing the clause learning process of a CDCL SAT solver, it is possible to conclude that learnt clauses need not contain auxiliary variables. Hence, the SAT solver can effectively only consider non-auxiliary variables for branching and clause learning purposes.

In practice, branching only on non-auxiliary variables may not be the most effective strategy. As a result, one challenge is to evaluate which auxiliary variables can be deemed more effective for branching purposes.

Proposition 8. *For the CNF encoding $LT^{n,1}_{SEQ}$ of constraint $\leq 1\ (x_1, \ldots, x_n)$, and such that all auxiliary variables are unassigned, then the assignments $s_0 = 1$ and the assignment $s_{n-1} = 0$ originate the largest number of implied assignments, or a conflict is identified.*

Proof. Assume a conflict is not identified. A simple inductive argument suffices. $s_1 = 1$ implies $s_2 = 1$, and the same holds true for all i, $1 \leq i \leq n-2$, $s_i = 1$ implies $s_{i+1} = 1$. In addition, all variables x_i, $1 < i \leq n$, are assigned value 0.
Similarly, $s_{n-1} = 0$ implies $s_{n-2} = 0$, and the same holds true for all i, $2 \leq i \leq n-1$, $s_i = 0$ implies $s_{i-1} = 0$. In contrast with the previous case, all variables x_i, $1 \leq i < n$, are assigned value 0.
Clearly, all the other auxiliary variables yield no more implied assignments than these auxiliary variables. □

Hence, a possible approach for deciding which auxiliary variables to consider for branching purposes is to select the first and last auxiliary variables of each $LT^{n,1}_{SEQ}$ encoding, the first variable is preferred to be assigned value 1, and the last is preferred to be assigned value 0.

Motivated by the previous results, a number of possible variable branching heuristics can thus be devised:

1. Branch on all variables, both non-auxiliary and auxiliary. This branching heuristic will be implemented by any existing CDCL SAT solver.
2. Branch only on non-auxiliary variables. This branching heuristic attempts to replicate the branching steps in the pairwise encoding.
3. Branch only on non-auxiliary variables, and on the auxiliary variables for which one of the value assignments guarantees the largest number of implied assignments. The rationale is that if branching on some auxiliary variables is useful, then these should be the preferred variables to branch on.

In order to implement the last two branching heuristics, a CDCL SAT solver needs to be adapted to accept branching directives from the CNF formula description. This can be achieved by specifying these directives as comments in the standard input format for SAT solvers. Section 5 evaluates these branching heuristics.

4.2 Modifications to a CDCL SAT Solver

This section focus on CDCL SAT solvers, and shows how the SAT solver can be modified to allow filtering auxiliary variables for branching purposes, as outlined in the previous section. The MiniSat [9] SAT solver is assumed, since the proposed modifications are straightforward to implement in MiniSat. Regarding the three main engines of a CDCL SAT solver, the deduction engines requires no modification, the decision engine needs to filter variables not used for branching purposes, and the diagnosis engine needs to exchange learnt literals on non-branching variables by literals on branching variables. A more detailed description of the modifications to the CDCL SAT solver is given below.

Internal Data Structures. When creating new variables, the SAT solver is informed of whether a variable is non-auxiliary, and so needs to be considered for variable branching purposes, or whether it is auxiliary, and so needs not be considered for variable branching purposes. For the auxiliary variables, the first and last auxiliary variables of each $LT^{n,1}_{SEQ}$ encoding can also be optionally considered for branching purposes. In addition to specifying the variables that can serve for branching purposes, with each such variable the preferred value can also be specified. In these cases, the preferred value is always used when the SAT solver branches on that variable.

The Decision Engine. When selecting a variable for branching, the SAT solver is modified to only consider variables that can serve as branching variables. Hence, the decision engine only branches on variables that were initially declared to be eligible as branching variables. Moreover, if a preferred value is associated with a given variable, then the decision engine uses the preferred value when branching on that variable.

The Diagnosis Engine. When learning a conflict clause, if the UIP condition holds and if the current variable is auxiliary, then the clause learning process continues. The results of the previous section (see Propositions 6 and 7) guarantee that the UIP condition will remain valid until a non-auxiliary UIP variable is identified or until a auxiliary branching variable is identified. As shown earlier, the size of learnt clauses is unchanged by filtering auxiliary variables.

5 Experimental Results

The ideas described above have been implemented in the most recent version of the MiniSat SAT solver [9] - MiniSat2, a cleaned up version of the winning entry of SAT-Race 2006. MiniSat is a CDCL solver, containing all the features of the current state-of-the-art solvers: conflict-clause learning, conflict-driven backjumping, dynamic variable ordering heuristic and two watched-literal scheme.

We have considered two different encodings:

1. The *pairwise encoding* (`pw`), which has a quadratic number of clauses but no additional variables.
2. The *sequential counter encoding* (`sc`), representing $LT_{SEQ}^{n,1}$, which has a linear number of clauses and also a linear number of auxiliary variables.

The sequential counter encoding has been evaluated for two additional configurations of the MiniSat SAT solver:

1. `sc-d`: for this configuration the *decision* variables are selected from the non-auxiliary variables only (and therefore auxiliary variables cannot be selected).
2. `sc-dh`: for this configuration the decision variables are selected from the non-auxiliary variables and also from two auxiliary variables: s_1 and s_{n-1} (again the remaining auxiliary variables cannot be selected). In case any of these two auxiliary variables are selected, *hints* are given for the value to assign to these variables: if variable s_1 is selected then it is assigned value 1, and if variable s_{n-1} is selected then it is assigned value 0. These assignments originate the largest number of implied assignments.

For the results given below, the main goal is to evaluate (1) the performance of the pairwise encoding against the sequential encoding in terms of the CPU time and the memory required and (2) the improvements achieved by the filtering of auxiliary variables. All the results were obtained on an Intel Xeon 5160 (3.0GHz with 4GB of RAM) and a timeout (TO) of 1000s.

5.1 Problem Instances

A number of problems were evaluated, all containing many ≤ 1 $(x_1, \ldots, x_n)$ constraints. These problems were the following: the n-queens problem, the pigeon hole problem, the round-robin problem, the all-interval series problem, the graph coloring problem and the Latin squares problem extended with constraints on (broken) diagonals [4]. The analysis of results is divided into two classes: instances for the n-queens problem, which modern SAT solvers can tackle, and instances from the other problems considered, since for these problems modern SAT solvers can solve only a few instances.

5.2 Results for the N-Queens Problem

The n-queens problem is the problem of placing n chess queens on an $n \times n$ chessboard such that no two queens share the same row, column, or diagonal, i.e. there is at most one queen in each row, column or diagonal. We may represent this problem with $n \times n$ Boolean variables, where each variable corresponds to

[4] The quasigroup completion problem was also evaluated near the phase transition. For this problem the number of literals in the sequential encoding is in general larger than the number of literals in the pairwise encoding. Given that some entries in the quasigroup are already defined, the number of entries to be distinct is reduced.

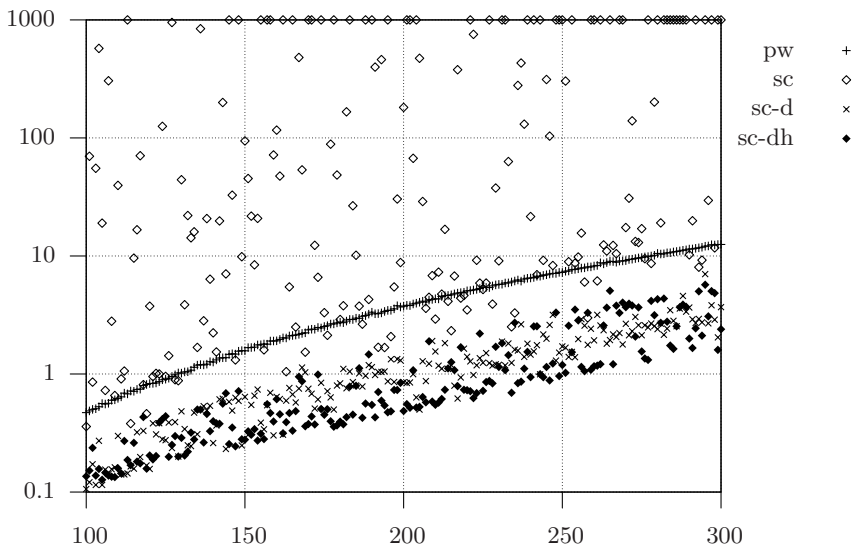

Fig. 1. Results on run times.

one entry in the chess board. We then require $2 \times n$ constraints ≤ 1 $(x_1, \ldots, x_n)$ to guarantee that there is only one queen per row and one queen per column. In addition, we require $4n - 6$ constraints to guarantee that there is at most one queen per diagonal. We have generated n-queens problems ranging from $n = 100$ to $n = 300$ using both the pairwise (`pw`) and the sequential counter (`sc`) encoding. CNF formulas generated by both encodings were solved using the MiniSat SAT solver. In addition, two modified versions of MiniSat (`sc-d` and `sc-dh`) were evaluated for the sequential encoding.

Figure 1 gives the CPU time (in seconds) for solving the n-queens problems using the four different approaches. From this figure a few conclusions can be drawn:

- The sequential encoding, when used with the plain MiniSat solver (`sc`) is quite unstable and in general takes more time than any other approach. Also, it is not able to solve 51 problem instances within the allowed CPU time (1000s).
- The pairwise encoding (`pw`) although being stable requires in general up to one order of magnitude more time than the two other approaches using the sequential encoding (`sc-d` and `sc-dh`).
- Both `sc-d` and `sc-dh` are more competitive than the `pw` approach. This contrasts with the `sc` approach. Not only the size of the search space in `sc` is larger but also the search *gets lost* recording useless clauses and being unable to find a solution.

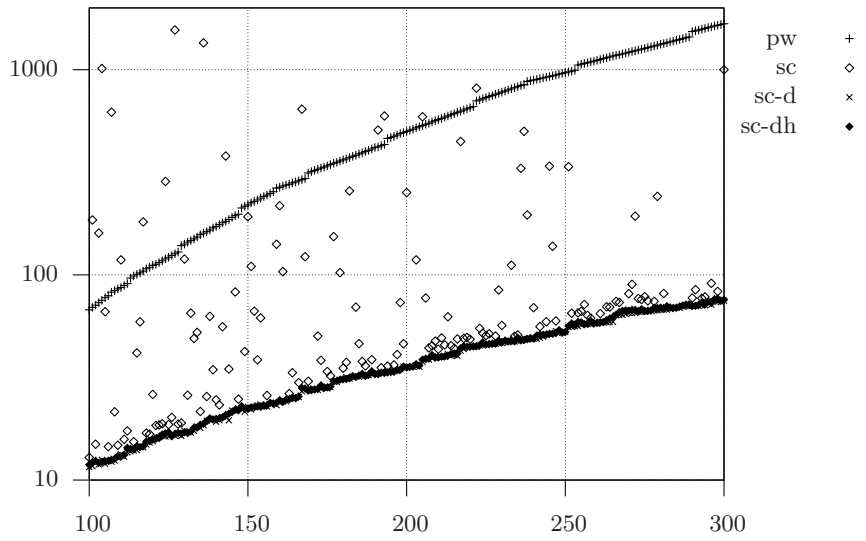

Fig. 2. Results on memory used.

Figure 2 evaluates the memory consumed by MiniSat (in MB) for solving a given problem instance following one of the four approaches. Again, from this figure a few conclusions can be drawn:

- The pairwise encoding (`pw`) requires more memory than both the `sc-d` and `sc-dh` approaches. The memory difference most often exceeds one order of magnitude. This difference is significant in practice: the `pw` requires around 1600 MB for the larger problem instances, whereas 75 MB suffice for the `sc-d` and `sc-dh` approaches.
- Although the CNF formula is exactly the same for `sc`, `sc-d` and `sc-dh`, the amount of memory required by `sc` is significantly larger. This is due to the clauses recorded during the search, resulting from the conflicts.

Figure 3 gives the number of decisions, corresponding to the number of nodes in the search tree, required by each of the approaches for solving a given problem instance.

- The `sc` approach explores by far the largest number of nodes. This is in part due to the larger search space. Moreover, it is also clear that the heuristic used is far from being accurate for this encoding.
- Both `sc-d` and `pw` approaches explore the smallest number of nodes. Given that `sc-d` only branches on non-auxiliary variables, both approaches may explore the same search space. This means that each one of the heuristics used by each one of the approaches is very effective for that specific approach.
- The `sc-dh` approach explores more nodes than `sc-d` and `pw`. This comes as no surprise given that the search space is larger. For each $\leq 1\ (x_1, \ldots, x_n)$ constraint we may have two more decision variables. However, given the

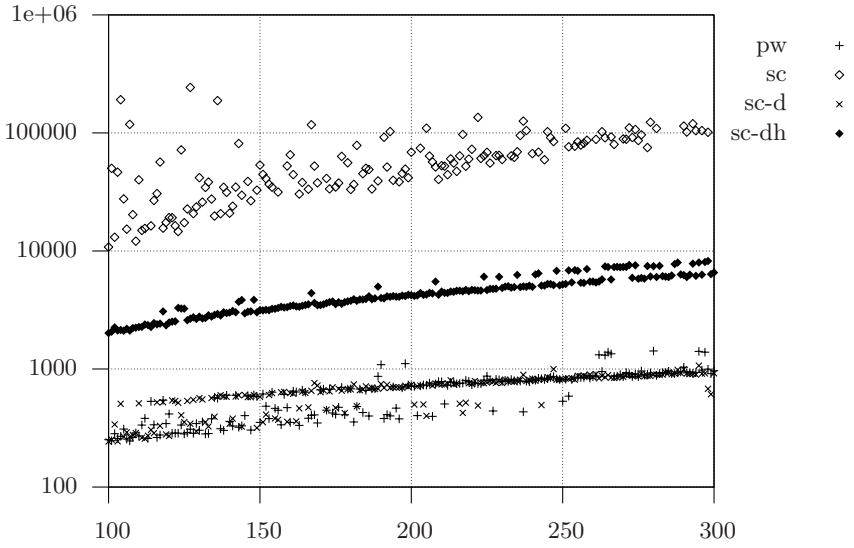

Fig. 3. Results for the number of decisions.

CPU times, these variables when taken as decisions are not as harmful as the other auxiliary variables when taken as decisions in the `sc` approach. This is probably due to the *hints* on the values to be assigned to the auxiliary variables.

Finally, figure 4 reveals the number of conflicts that are found during the search. This figure further clarifies the differences between the `sc-d` and the `sc-dh` approaches. Interestingly, `sc-dh` has the smallest number of conflicts, even though the number of decision nodes is significant, as shown in the previous figure. Our interpretation is that many decisions are irrelevant, but a few are extremely useful. This makes the conflict clauses in general shorter and therefore able to prune more effectively the search space.

5.3 Results for Other Problems

This section presents results for a number of other instances which are also encoded with $\leq 1\ (x_1, \ldots, x_n)$ constraints. Example instances from the all-interval series, pigeon-hole, Latin squares and round-robin problems are considered. For these problems only a few instances are considered. Other instances are not considered, either because run times are negligible or because modern SAT solvers exceed the allowed CPU time limit.

The results are summarized in Table 5.3. As observed for the n-queens problem, the approaches `sc`, `sc-dh` and `sc-d` tend to perform more robustly than the `pw` approach, with a few outliers. In terms of memory used, no concrete pattern was identified in the results, in part because of the small number of instances

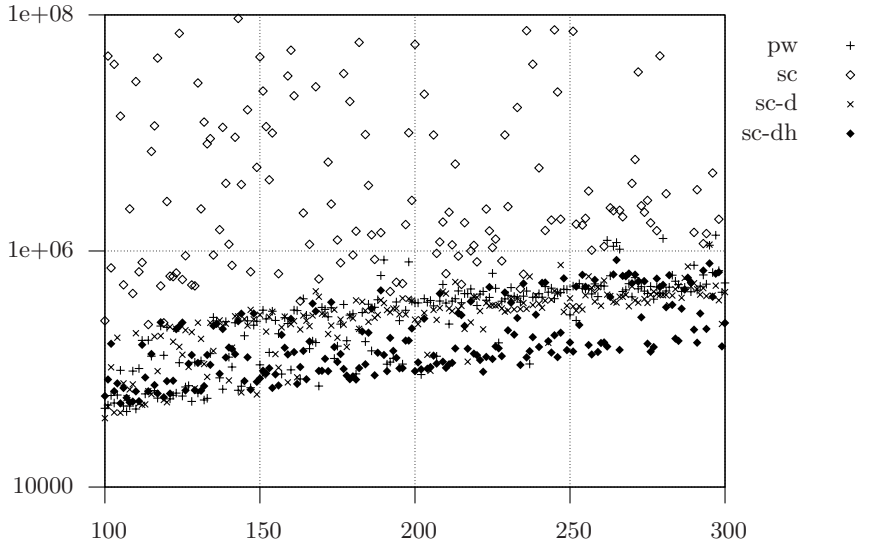

Fig. 4. Results for the number of conflicts.

that can be considered. For instances with similar run times, the `sc` encodings use significantly less memory than the `pw` encoding (e.g. *rr10* and *php10*).

6 Conclusions

This paper studies techniques for improving the robustness of linear size encodings of ≤ 1 $(x_1, \ldots, x_n)$ constraints. The sequential counter ($LT^{n,1}_{SEQ}$) encoding of [21] is considered. For this encoding, the additional auxiliary variables used for encoding ≤ 1 $(x_1, \ldots, x_n)$ constraints can essentially be ignored by the SAT solver. A related result is that auxiliary variables can be discarded from learnt clauses, without affecting the number of literals of each learnt clause. An additional result is that auxiliary variables are guaranteed to yield different numbers of implied variables, and so this yields a natural ranking of auxiliary variables for branching purposes.

A number of different strategies for selecting branching variables are outlined and experimentally evaluated. Experimental results indicate that filtering most of the auxiliary variables is an effective technique, yielding significantly more robust CDCL SAT solvers, and may represent a valid alternative to the space-consuming pairwise encoding.

A future line of research is to devise techniques for extending the work in this paper, by being more precise at filtering auxiliary variables that are not effective for branching purposes. The properties identified for the $LT^{n,1}_{SEQ}$ encoding also suggest a family of branching heuristics, besides the ones outlined in the paper. These heuristics allow increasing number of decision variables ranked

Table 1. Results on additional instances.

Bench	pw		sc		sc-d		sc-h	
	mem	time	mem	time	mem	time	mem	time
ais16	6.74	5.12	3.68	0	8.06	45.12	7.29	24.57
ais17	13.27	77.33	3.8	0	6.51	8.24	9.28	53.95
ais18	—	TO	15.57	550.02	18.56	860.78	16.12	401.24
php9	3.54	2.4	3.54	1.6	3.54	4.76	3.54	4.18
php10	4.06	31.73	3.66	15.74	3.79	43.26	3.78	45.75
php11	5.45	526.03	4.41	189.51	4.52	652.91	4.53	741.6
ls8	7.45	40.51	7.31	64.58	6.71	33.51	6.61	34.56
rr10	17.28	1.03	13.11	1.83	12.91	1.74	8.77	0.39
rr12	41.23	4.28	90.85	50.26	20.21	3.71	106.79	50.79
gc-anna	9.93	18.87	10.71	39.98	9.98	27.56	10.52	42.05
gc-david	9.47	29.66	9.3	30.64	9.59	41.64	9.41	33.17
gc-huck	8.77	32.71	7.66	25.01	8.23	33.14	8.48	30.69

by the number of guaranteed implied assignments. One possible application of these heuristics would be to instruct SAT solvers to switch between branching heuristics after each search restart.

Finally, another line of research is to extend the results in the paper to other cardinality constraints. Results equivalent to the ones proposed in this paper, namely Proposition 2, are expected to exist for most linear encodings of constraint $\leq 1\ (x_1, \ldots, x_n)$, and for encodings of general cardinality constraints $\leq k\ (x_1, \ldots, x_n)$. A more challenging question is how some of the other results proposed in the paper, namely the ones related with clause learning, can be adapted either to other encodings of constraint $\leq 1\ (x_1, \ldots, x_n)$ or to general cardinality constraints $\leq k\ (x_1, \ldots, x_n)$.

Acknowledgments This work is partially supported by Fundação para a Ciência e Tecnologia under research projects POSC/EIA/61852/2004 and POSI/SRI/41926/01, EPSRC grant EP/E012973/1, and EU project IST/033709.

References

1. C. Ansótegui and F. Manyá. Mapping problems with finite-domain variables to problems with boolean variables. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 1–15, 2004.
2. O. Bailleux and Y. Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 108–122, 2003.
3. O. Bailleux and Y. Boufkhad. Full CNF encoding: The counting constraints case. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2004.
4. O. Bailleux, Y. Boufkhad, and O. Roussel. A translation of pseudo Boolean constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2, March 2006.
5. R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, pages 203–208, July 1997.

6. R. Béjar, R. Hähnle, and F. Manyà. A modular reduction of regular logic to classical logic. In *Proceedings of the International Symposium on Multiple-Valued Logics*, pages 221–226, 2001.
7. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, July 1962.
8. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, July 1960.
9. N. Eén and N. Sörensson. An extensible SAT solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, May 2003.
10. N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2, March 2006.
11. I. P. Gent. Arc consistency in SAT. In *Proceedings of the European Conference on Artificial Intelligence*, pages 121–125, 2002.
12. I. P. Gent and P. Nightingale. A new encoding of AllDifferent into SAT. In *Proceedings 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 95–110, September 2004.
13. I. P. Gent and P. Prosser. An empirical study of the stable marriage problem with ties and incomplete lists. In *Proceedings of the European Conference on Artificial Intelligence*, pages 141–145, 2002.
14. E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Proceedings of the Design and Test in Europe Conference*, pages 142–149, March 2002.
15. C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the National Conference on Artificial Intelligence*, pages 431–437, July 1998.
16. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990.
17. J. Marques-Silva and K. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, November 1996.
18. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, June 2001.
19. A. Nadel. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Hebrew University of Jerusalem, November 2002.
20. L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, February 2004.
21. C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 827–831, October 2005.
22. T. Walsh. SAT *v* CSP. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 441–456, September 2000.
23. J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.
24. H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997.
25. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285, 2001.