Exploiting Cardinality Encodings in Parallel Maximum Satisfiability

Ruben Martins, Vasco Manquinho and Inês Lynce
INESC-ID/IST, TU Lisbon, Portugal
{*ruben,vmm,ines*}*@sat.inesc-id.pt*

***Abstract*—Cardinality constraints appear in many practical problems and have been well studied in the past. There are many CNF encodings for cardinality constraints, although it is not clear which encodings perform better. Indeed, different encodings can perform well over different problems. This paper examines a large number of cardinality encodings and evaluates their performance for solving the problem of Maximum Satisfiability (MaxSAT). Taking advantage of the diversification of cardinality encodings, we propose to exploit those encodings in parallel MaxSAT solving. Our parallel solver, pMAX, simultaneously searches in the lower and upper bound of the optimum value, and different cardinality encodings are used in each thread to increase the diversification of the search. Moreover, learned clauses are shared between threads during the search. Experimental results show that our parallel solver outperforms other sequential and parallel state-of-the-art MaxSAT solvers.**

I. Introduction

Cardinality constraints appear in many practical problems and can also be used in the context of Maximum Satisfiability (MaxSAT) solving. There are several encodings that translate cardinality constraints into clauses and these encodings have many practical applications. For example, a translation is necessary when we want to use a Boolean Satisfiability (SAT) or MaxSAT solver that is not able to handle natively cardinality constraints. Even though there are many encodings, it is not clear which encoding should be used when facing a cardinality constraint. This paper proposes to examine several cardinality encodings and to perform an experimental evaluation of the encodings for MaxSAT solving.

An increasing number of parallel SAT solvers have emerged in the last years as a result of the predominance of multicore processors. Even though parallel approaches are known to boost performance, in the context of MaxSAT these approaches are scarce. Therefore, the diversification of cardinality encodings can be seen as a starting point for the design of a new parallel MaxSAT solver. By exploiting the different cardinality encodings, we can build a portfolio of encodings to be used in parallel algorithms that search on the lower and upper bounds of the optimal solution. Moreover, learned clauses can be shared during search which further boosts the performance of the new parallel MaxSAT solver.

The main contribution of this paper is the development of a new parallel MaxSAT solver that uses a portfolio of cardinality encodings. Additionally, an empirical evaluation of a large number of cardinality encodings for MaxSAT solving is also presented.

The paper is organized as follows. The next section introduces some background notions of MaxSAT and cardinality constraints. Section III briefly resumes the several cardinality encodings that will be used in this paper. Next, section IV presents our parallel MaxSAT solver that uses a portfolio of cardinality encodings. The clause sharing mechanism and the integration of learned clauses during the search are also presented in this section. Section V describes related work in parallel MaxSAT solving. Next, section VI presents the experimental results that evaluate the different cardinality encodings and show that our parallel MaxSAT solver, pMAX, can outperform other sequential and parallel state-of-the-art MaxSAT solvers. Finally, section VII concludes the paper and suggests future work.

II. Preliminaries

In this section we briefly describe the MaxSAT problem and its variants, as well as some other definitions used throughout the remainder of the paper.

In a propositional formula, a literal l_j denotes either a variable x_j or its complement $\bar{x}_j$. If a literal $l_j = x_j$ and x_j is assigned value 1 or $l_j = \bar{x}_j$ and x_j is assigned value 0, then the literal is said to be *true*. Otherwise, the literal is said to be *false*. A propositional clause can be defined as a disjunction of literals and a CNF formula is a conjunction of propositional clauses. A clause is said to be unsatisfied if all its literals are assigned value 0, and it is said to be satisfied if at least one of its literals is assigned value 1.

Given a CNF formula φ, the MaxSAT problem can be defined as finding an assignment to problem variables such that it minimizes (maximizes) the number of unsatisfied (satisfied) clauses. MaxSAT has several variants such as partial MaxSAT, weighted MaxSAT and weighted partial MaxSAT. In the partial MaxSAT problem, some clauses in φ are declared as hard, while the rest are declared as soft. The objective in partial MaxSAT is to find an assignment to problem variables such that all hard clauses are satisfied, while minimizing the number of unsatisfied soft clauses. Finally, in the weighted versions of MaxSAT, soft clauses can have weights greater than 1 and the objective is to satisfy all hard clauses while minimizing the total weight of unsatisfied soft clauses.

A generalization of clauses are cardinality constraints. These constraints define that a sum of n literals must be smaller than or equal to a given value k, i.e. $\sum_{i=1}^{n} l_i \leq k$. Although cardinality constraints do not occur in MaxSAT formulations, several algorithms for MaxSAT solving rely on these constraints. For example, they are used in unsatisfiability-based algorithms [13], [19], [1] and in partial MaxSAT algorithms that perform linear search on the number of unsatisfiable soft clauses [18].

TABLE I
ENCODINGS FOR CARDINALITY CONSTRAINTS

Encoding	Clauses	Variables	Type
Pairwise	$\mathcal{O}(n^2)$	0	*at-most-one*
Ladder	$\mathcal{O}(n)$	$\mathcal{O}(n)$	*at-most-one*
Bitwise	$\mathcal{O}(n\ log_2\ n)$	$\mathcal{O}(log_2\ n)$	*at-most-one*
Commander	$\mathcal{O}(n)$	$\mathcal{O}(n)$	*at-most-one*
Product	$\mathcal{O}(n)$	$\mathcal{O}(n)$	*at-most-one*
Sequential	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	*at-most-k*
Totalizer	$\mathcal{O}(nk)$	$\mathcal{O}(n\ log_2\ n)$	*at-most-k*
Sorters	$\mathcal{O}(n\ log_2^2\ n)$	$\mathcal{O}(n\ log_2^2\ n)$	*at-most-k*

In both of these approaches, new cardinality constraints are iteratively added to the original formula. Hence, in order to continue using a SAT solver in the subsequent iterations, it is necessary to encode cardinality constraints into clauses. Another option is to use a pseudo-Boolean satisfiability solver that is able to deal natively with cardinality constraints.

III. ENCODINGS FOR CARDINALITY CONSTRAINTS

Due to the practical importance of cardinality constraints, their encoding into CNF has been the subject of several studies in the last decade. In this section is provided a description of several encodings that will be used in the remainder of the paper.

Our focus is on cardinality encodings that allow unit propagation to maintain arc consistency in the resulting CNF encoding. Consider the following cardinality constraint $x_1 + \ldots + x_n \leq k$. If k variables are assigned value *true*, then unit propagation will enforce the value *false* on the remaining $n-k$ variables. However, if $k+1$ variables are assigned value *true*, then a conflict arises since at most k variables can be assigned value *true*. An encoding that maintains arc consistency enables the SAT solver to infer the same information with the use of unit propagation on the resulting CNF encoding.

A special case of cardinality constraints are the *at-most-one* constraints. These constraints express that at most one out of n Boolean variables is allowed to be *true*. A large number of encodings have been proposed to handle *at-most-one* constraints. Therefore, a distinction is made between encodings that are used only for *at-most-one* constraints and the encodings used for the general case of *at-most-k* constraints.

Table I shows the size of the evaluated encodings. Next, the encodings are briefly described. Due to lack of space, not much detail is given for each of the encodings and the reader is pointed to the literature.

- Pairwise (naive): the most widely known encoding for the *at-most-one* constraint. For each pair of variables (x_i, x_j), we add a binary clause $\bar{x}_i \vee \bar{x}_j$ that guarantees that only one of these two variables can be assigned value *true*. Even though this encoding adds a quadratic number of clauses, it does not require auxiliary variables.
- Ladder [14], [2]: it uses $n-1$ auxiliary variables to form a structure denoted by ladder. Consider the chain of auxiliary variables $y_1, \ldots, y_{n+1}$. If y_i is *false* then all variables y_j with $j > i$ are also *false*. Each valid state in the ladder represented by $(y_i \wedge \bar{y}_{i+1})$ is associated with a variable of the cardinality constraint. Since each x_i is equivalent to a valid state in the ladder, this encoding guarantees that at most one x_i will have value *true*.
- Bitwise [12], [23]: this encoding introduces auxiliary variables $y_1, \ldots, y_{log_2\ n}$ that represent a bit string. It then associates a unique bit string with each x_i. The encoding guarantees that only one string can occur and therefore at most one variable x_i can have value *true*. When n is not a power of 2, we can perform a small optimization by reducing the number of clauses from the encoding [12]. If n is not a power of 2, then there are more strings than variables x_i. Hence, we can associate two strings to some of the x_i until the number of remaining strings is equal to the number of remaining x_i variables.
- Commander [17]: it starts by partitioning the set of variables x_i into groups of size 3. Next, for the variables of each group, an *at-most-one* constraint is encoded with the pairwise encoding. Finally, it associates a commander variable with each group and it recursively encodes the *at-most-one* constraint over the commander variables with the method just described.
- Product [8]: this encoding decomposes cardinality constraint $x_1 + \ldots + x_n \leq 1$ into two constraints, $y_1 + \ldots + y_{p_1} \leq 1$ and $z_1 + \ldots + z_{p_2} \leq 1$, where $p_1 \times p_2 \geq n$. The idea is to associate each x_i with a coordinate (y_a, z_b). This procedure is applied recursively until the size of the constraint is smaller than 7. At that point, the pairwise encoding is used.
- Sequential [24]: it encodes a circuit that sequentially counts the number of variables x_i that are assigned value *true*. Each x_i is associated with k variables $s_{i,j}$ that are used as a counter. Assigning the value *false* to $s_{i,j}$ implies that at most j of the $x_1, \ldots, x_{i-1}$ variables can have value *true*.
- Totalizer [5]: it consists of a totalizer and a comparator. The totalizer can be seen as a binary tree, where the leaves are the x_i variables. Each intermediate node is labeled with a number s and uses s auxiliary variables to represent the sum of the leaves of the corresponding subtree. The original encoding uses $\mathcal{O}(n^2)$ clauses. However, since we are using this encoding to encode *at-most-one* constraints, the optimization proposed by Büttner and Rintanen [7] that reduces the number of clauses to $\mathcal{O}(nk)$ is used. The idea is that instead of counting up to n, it is enough to count up to $k+1$, which can be used to reduce the number of variables used in each node.
- Sorters [10]: it is based on a sorting network, i.e. a circuit that receives n Boolean inputs $x_1, \ldots, x_n$ and permutes them to obtain the sorted outputs $y_1, \ldots, y_n$. Consider the cardinality constraint $x_1 + \ldots + x_n \leq k$. If after building the sorting network we assign *false* to the output y_{k+1}, this guarantees that at most k variables x_i can have value *true*. Some improvements were introduced over the original sorting network encoding, namely, the use of half sorting networks [3] and additional redundant clauses over the outputs that amplify propagation [9]. Moreover,

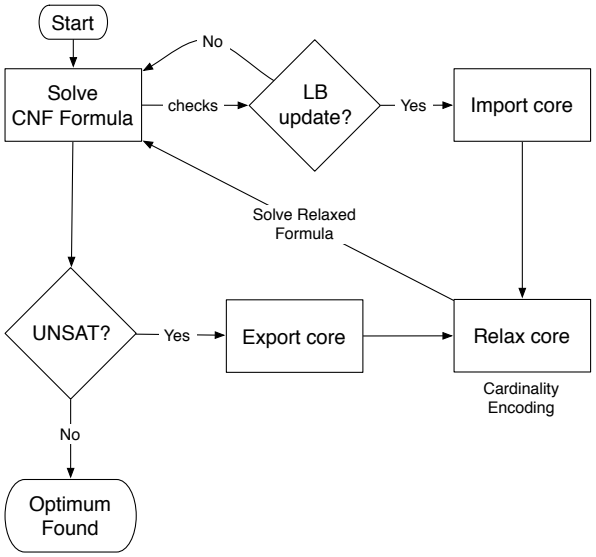


Fig. 1. Parallel Unsatisfiability-based Algorithms

for the *at-most-one* constraints, the simplification of the sorting network through partial evaluation [9] is used and the size of the encoding is reduced to $\mathcal{O}(n)$ clauses and variables.

IV. Exploiting Cardinality Encodings

When using a pseudo-Boolean (PB) solver, cardinality constraints can be used natively without the need to encode them into CNF. However, encoding cardinality constraints into CNF can outperform the PB representation for some classes of problems. Moreover, different encodings may have distinct performance in various problems.

In a multicore architecture, one can exploit the variety in the performance of cardinality encodings by using a portfolio of algorithms using different encodings. The use of different cardinality encodings increases the diversification of the search. Moreover, these algorithms can cooperate with each other by exchanging learned clauses and information about the optimum value. Next, the architecture of our parallel solver and the clause sharing mechanism is described.

A. Solver Architecture

Our parallel MaxSAT solver is based on two orthogonal algorithms: (1) unsatisfiability-based algorithms that search on the lower bound of the optimal solution and (2) linear search algorithms that search on the upper bound. Therefore, we propose to perform a parallel search on both sides of the optimum solution. Moreover, to increase the diversification of the search, different cardinality encodings for each algorithm will be used.

Figure 1 shows an overview of parallel unsatisfiability-based algorithms. These algorithms work by iteratively identifying unsatisfiable sub-formulas of the original formula. While solving the formula, the algorithm checks if another thread has found a better lower bound value, i.e. if it has found an unsatisfiable sub-formula. If this is the case, it imports the unsatisfiable sub-formula (core) and performs core relaxation. For each soft clause in the identified unsatisfiable sub-formula, a new relaxation variable is added such that when this variable is assigned to 1, the soft clause becomes satisfiable [13]. Moreover, a *cardinality constraint* is also added to the relaxed formula such that only one of the newly created relaxation variables can be assigned value 1. Next, the solver checks if the formula remains unsatisfiable.

If the algorithm is not informed that a better lower bound value was found, it continues the search process until it finds an unsatisfiable sub-formula or a solution to the formula. If it finds an unsatisfiable sub-formula, it shares this unsatisfiable core with the remaining lower bound threads by exporting the core to the other threads. Next, it relaxes the unsatisfied sub-formula as previously described and continues the search on the relaxed formula.

The procedure ends when the working formula becomes satisfiable and the solver returns a solution (i.e. the optimum value was found), or if the unsatisfiable sub-formula only contains hard clauses (i.e. the original problem instance is unsatisfiable) [13].

To increase the diversification of the search, unsatisfiability-based algorithms use different cardinality encodings in the relaxation step. These cardinality constraints are *at-most-one* constraints and any of the *at-most-one* cardinality encodings presented in the previous section can be used.

There are a few details not shown in figure 1. In particular, exporting a core is inside a critical region and locks are used to avoid two or more threads of exporting a core that would correspond to the same lower bound value. Therefore, before exporting a core, a thread checks if its lower bound value is the highest lower bound value among all threads. If this is the case, then it is safe to export the core to the remaining threads. Otherwise, it discards its own core and imports the core that corresponds to the current lower bound value. This is done to guarantee that all threads use the same cores. Moreover, when a thread relaxes a core, it updates its lower bound value.

Figure 2 shows an overview of parallel linear search algorithms. Notice that the original MaxSAT formula φ_{MS} is modified by adding a new relaxation variable r_i to each soft clause ω_i from φ_{MS}, resulting in an equivalent formulation φ_{UB} where one wants to minimize the number of relaxation variables assigned value 1. In this approach, whenever a new solution is found for φ_{UB}, the upper bound value is updated and a new *cardinality constraint* on the relaxation variables is added such that all solutions with a greater or equal value are excluded. During search, each algorithm checks if there is a better upper bound value. If this is the case, it adds a cardinality constraint considering the new upper bound value. Afterwards, it restarts the search on the constrained formula.

To increase the diversification of the search, the linear search algorithms to be used differ between themselves on the cardinality encoding that is used when a new upper bound value is found. As a result of finding a new upper bound of value k', the cardinality constraint $x_1 + \ldots + x_n \leq k$ becomes increasingly stronger by decreasing k to k'. For the *at-most-k*

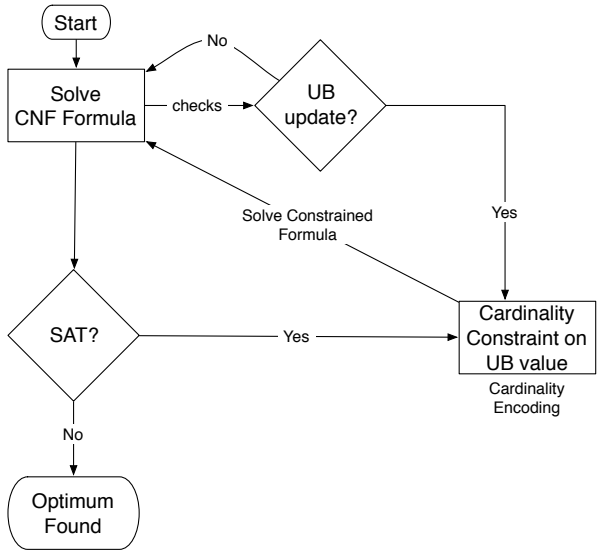


Fig. 2. Parallel Linear Search Algorithms

encodings presented in the previous section it is only needed to encode the cardinality constraint into CNF when the first upper bound value is found. In the next iterations, one can set some specific literals in the encoding to *false* such that it restricts the cardinality constraint to the new upper bound value. This is denoted by *incremental strengthening*. All learned clauses from previous iterations remain valid and can therefore be kept.

Since our parallel solver runs both unsatisfiability-based algorithms and linear search algorithms, they can cooperate by exchanging information about the lower and upper bound values. If during the search, the lower and upper bound values become the same, it means that the optimum solution has been found. Therefore, it is not necessary for any of the threads to continue the search to prove optimality since their combined information already proves it. Additionally, learned clauses can be shared among all algorithms thus reducing the search space of each algorithm. Next, the clause sharing procedure is explained in detail.

B. Clause Sharing

Similarly to parallel SAT solving, only learned clauses that have less than a given number of literals are shared among all threads. In our parallel solver, we start by sharing learned clauses that have 8 or fewer literals. This cutoff is dynamically changed using the throughput and quality heuristic proposed by Hamadi et al. [15]. Additionally, all clauses with literal block distance 2 are also shared [4].

However, in our parallel solver not all learned clauses can be shared among all threads. This is due to the fact that the working formulas are different. As previously explained, unsatisfiability-based algorithms work directly with the input formula φ_{MS}, while algorithms that perform a linear search on the upper bound value, use relaxation variables on the soft clauses, resulting in formula φ_{UB}.

In order to define the conditions for safe clause sharing, we start by remembering the definition of soft and hard learned clauses [22]. If the conflict which gave origin to a new clause only involves hard clauses, then the learned clause is said to be a hard learned clause. Otherwise, it is said to be a soft learned clause. We refer to the literature for a detailed explanation of the clause learning procedure [21], [25].

Since φ_{MS} contains both soft and hard clauses, it will also have soft and hard learned clauses. On the other hand, φ_{UB} only has hard clauses, and as a result will only have hard learned clauses. Nevertheless, as mentioned previously, φ_{UB} contains additional relaxation variables that are not present in φ_{MS}. When using cardinality encodings, we also have to take into account the auxiliary variables used by those encodings. Therefore, each thread may contain variables not present in the other threads. As a result, the safe sharing procedure between the different algorithms is as follows:

- Hard learned clauses from unsatisfiability-based algorithms that do not have auxiliary variables can be safely shared with the other threads.
- Soft learned clauses from unsatisfiability-based algorithms are not shared with the other threads. These clauses may not be valid for formulas φ_{UB} and cannot be shared with the algorithms that perform linear search on the upper bound.
  Notice that these clauses could be shared with other threads that are using unsatisfiability-based algorithms. However, it would be necessary to establish an equivalence between the relaxation variables of the learned soft clause and the relaxation variables of the importing thread. Since variables are created for the encoding of cardinality constraints, the identification of the relaxation variables may differ between threads. Hence, even though it would be possible to share soft learned clauses between unsatisfiability-based algorithms it is currently not implemented in our parallel solver.
- Hard learned clauses generated when solving φ_{UB} can be shared with the other threads if the learned clause does not contain relaxation or auxiliary variables.

Finally, between iterations of the unsatisfiability-based algorithms, the working formulas φ_{MS} are also extended with additional relaxation variables. However, since these variables are added to soft clauses, if a conflict-based learned clause contains any relaxation variable, then it will necessarily be considered a soft clause. This is due to the fact that at least one soft clause would have been used in the learning procedure.

C. Integration of Learned Clauses

Whenever a learned clause is generated, if its size is lower than the current cutoff, it is exported as a learned clause for the other threads. At the same time, when a thread checks if there is a better lower or upper bound value, it also imports the learned clauses that were shared by other threads. Since importing clauses is done during the search, the learned clauses have to be integrated in the context of the current search space. Therefore, for the integration of a shared clause ω we have to take into consideration the following cases:

- ω is a unit clause. A restart is forced and the corresponding literal is assigned.
- ω is unit in the current context. The SAT algorithm backtracks to the highest decision level of the variables in ω. A decision level of a variable denotes the depth of the decision tree at which the variable is assigned a value. After the backtrack, the unassigned literal is assigned and propagated.
- ω is unsatisfied in the current context. The SAT algorithm backtracks to the highest decision level of the variables in ω. Conflict analysis is performed to allow further backtracking. Moreover, during the conflict analysis procedure a new clause will be learned.
- ω is satisfied in the current context. If exactly one literal in ω is satisfied and the remaining literals are falsified, and if the decision level of the satisfied literal is higher than the decision levels of all falsified literals, then the algorithm backtracks to the highest decision level among the falsified literals.

In the remaining cases the learned clause is simply added to the importing thread and no backtracking is needed. The integration procedure must be done in order to ensure the correctness of the solver. A similar procedure is done in the parallel SAT solver `ManySAT` [16].

V. Related Work

Portfolio approaches explore the parallelism provided by different viewpoints on the same problem. In parallel SAT solving, portfolio approaches are becoming the most common approach [16], [6], as they explore the sensitivity to parameter tuning of SAT solvers. As a result, each thread has a different combination of parameters, for example, different restart strategies, decision heuristics or learning schemes.

Even though there is an increasing number of parallel SAT solvers, there are only a few parallel implementations for solving optimization problems.

`SAT4J PB RES//CP` [1] implements a resolution based algorithm that runs in parallel with a cutting plane based algorithm to find a new upper bound or to prove optimality. When one of the algorithms finds a new upper bound, it terminates the search of the other algorithm and both restart their search within the new upper bound. If one of the algorithms proves optimality, then the problem is solved and the search is stopped. Clauses are not shared between these two algorithms.

`pwbo` [22] is a parallel Boolean Optimization solver that searches on the lower and upper bound of the objective value. If more than two threads are used, the remaining threads search on different local upper bound values. The parallel search on different local upper values leads to constants updates on the lower and upper bound values, which results in reducing the search space. In addition, to increase diversification of the search, `pwbo` uses two threads on the global upper bound value. One thread encodes the cardinality constraint into CNF, whereas the other thread uses a pseudo-Boolean constraint. This version is called `pwbo 4T-CNF` and will be used in the experimental results for comparison purposes.

The success of this approach motivated the exploitation of cardinality encodings. Nevertheless, the use of cardinality encodings in `pwbo` was restricted to one thread, since the focus of `pwbo` was the reduction of the search space by splitting the interval between the lower and upper bound values. In this paper, we propose to extend the use of cardinality encodings to all threads and to have a portfolio approach. Clause sharing was performed in `pwbo` at each restart and it was restricted to clauses of size 5 or smaller. Since the integration of learned clauses was done at decision level 0, `pwbo` did not implement the learned clause integration procedure described in the previous section.

VI. Experimental Results

This section evaluates the different cardinality encodings in MaxSAT solving and their application to parallel MaxSAT algorithms. All experiments were run on the partial MaxSAT instances from the industrial category of the MaxSAT Evaluation 2010 [2], which correspond to a set of 497 instances. The evaluation was performed on two AMD Opteron 6172 processors (2.1 GHz with 64 GB of RAM) running Fedora Core 13 with a timeout of 1,800 seconds (wall clock time).

For the parallel solvers, results were obtained by running each solver three times on each instance. Similarly to what is done when analyzing randomized solvers, the median time was taken into account. This means that an instance must be solved by at least two of the three runs to be considered solved. We should note, however, that this measure is more conservative than the one used in the SAT Race 2008 [3] where solving an instance in one run suffices to consider it solved.

This section is organized as follows: first, we present an empirical evaluation of the cardinality encodings for the *at-most-one* and *at-most-k* constraints. The diversification of cardinality encodings is then exploited to build a portfolio of encodings that is used in a parallel MaxSAT solver. Next, we present our parallel MaxSAT solver and perform an extensive evaluation of its performance against other sequential and parallel state-of-the-art MaxSAT solvers.

A. Encoding Evaluation

The different cardinality encodings were implemented in `wbo` [19], [20]. In `wbo` the search is initially done by a pseudo-Boolean (PB) solver that performs a search on the upper bound side of the optimal solution. However, the use of the PB solver is limited to 10% of the time limit given to solve the instance. If the PB solver proves optimality within this time limit, the optimal solution has been found without having to search on the lower bound side. On the other hand, if the PB solver was not able to prove optimality within the given time limit, an unsatisfiability-based algorithm is used to search on the lower

[1] `http://www.satcompetition.org/PoS/presentations--pos/leberre.pdf`

[2] `http://www.maxsat.udl.cat/10/`

[3] `http://baldur.iti.uka.de/sat-race-2008/`

TABLE II
NUMBER OF INDUSTRIAL PARTIAL MAXSAT INSTANCES SOLVED BY THE DIFFERENT CARDINALITY ENCODINGS

Benchmark	#I	*at-most-one* — Lower Bound Search									*at-most-k* — Upper Bound Search			
		Pairwise	Ladder	Bitwise	Comm.	Product	Seq.	Totalizer	Sorters	PB	Seq.	Totalizer	Sorters	PB
bcp-fir	59	44	50	46	52	47	49	50	49	44	51	53	51	10
bcp-hipp-yRa1	55	21	22	21	21	22	21	23	20	20	38	40	42	18
bcp-msp	64	3	3	3	4	4	4	5	5	4	26	26	26	12
bcp-mtg	40	17	19	16	18	17	17	18	17	17	40	40	40	26
bcp-syn	74	34	35	35	35	35	34	34	34	34	32	32	32	21
CircuitTrace	4	0	1	1	1	1	1	1	1	0	4	4	4	4
Haplotype	6	5	5	5	5	5	5	5	5	5	0	5	5	0
pbo-mqc	168	46	44	35	37	36	38	39	36	47	152	151	155	168
pbo-routing	15	15	15	15	15	15	15	15	15	15	15	15	15	13
PROTEIN_INS	12	1	1	1	1	1	1	1	1	1	2	2	2	1
Total	497	186	195	178	189	183	185	191	183	187	360	368	372	273

bound side. In the previous sections, we have seen that the cardinality constraint *at-most-one* is used in the lower bound search, whereas the cardinality constraint *at-most-k* is used in the upper bound search. To evaluate these two types of cardinality constraints, we have run `wbo` using only the lower bound search or the upper bound search.

Table II shows the number of instances solved when using different cardinality encodings. Since `wbo` can handle PB constraints natively, we also considered this representation of the cardinality constraints, i.e. without encoding them into CNF. The first column of table II shows the different set of benchmarks. The second column shows the number of instances per benchmark set. From column 3 to column 11 we can see the number of solved instances using solely lower bound search with the different *at-most-one* encodings. Similarly, in columns 12 to 15 we present the number of solved instances with upper bound search using different *at-most-k* encodings.

For the *at-most-one* cardinality encodings, the ladder encoding performed better overall. However, for most benchmarks the number of solved instances by each encoding is different. Moreover, there is no clear winner for all the benchmarks. This shows that cardinality encodings can diversify the search, since each encoding enables solving different instances. When the number of variables in the *at-most-one* constraint is small (less than a few hundred) then it is better to use the pairwise encoding or a PB representation. This occurs, for example, in the `pbo-mqc` benchmark. On the other hand, if the number of variables in the *at-most-one* constraint is large (thousands) then it is better to encode the constraint into CNF. This can be observed in the `bcp-fir` benchmark instances.

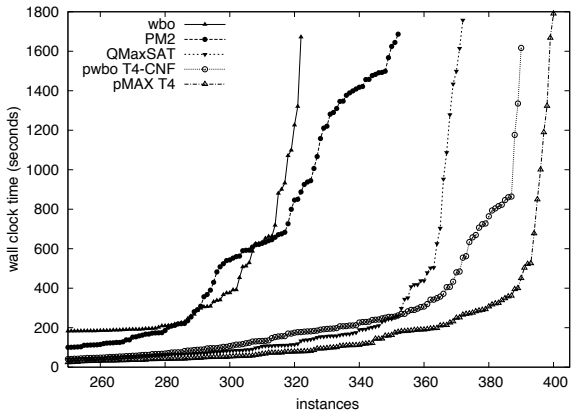
For the *at-most-k* cardinality encodings, the sorter network performed better overall. Even though the diversity of cardinality encodings for *at-most-k* is lower than for *at-most-one*, it can still show that different encodings may solve different instances. Encoding cardinality constraints into CNF is more effective when the number of variables is high (thousands) and the value of k is small when compared with the number of variables. This occurs, for example, in the `bcp-fir` benchmark. On the other hand, when given a cardinality constraint of size n with k close to $n/2$, using a PB representation can be more effective than encoding the cardinality constraint into CNF. This is the case of the `pbo-mqc` benchmark set.

To exploit the diversity showed by the cardinality encodings we propose to build a parallel solver as described in section IV. The new solver is called `pMAX` and can use 4 or 8 threads. To maintain a balance between lower and upper bound search, `pMAX` always uses the same number of threads for both approaches. `pMAX` uses a portfolio of cardinality encodings for algorithms that search on the lower and upper bounds. To build a portfolio of encodings for 4 and 8 threads we analyze table II and for each benchmark we tried to maximize the number of solved instances by our portfolio of encodings for the *at-most-one* and *at-most-k* constraints. Note that the fact that one encoding does not solve more instances than the others does not imply not using it, since it may solve many instances that were not solved when using other encodings.

With 4 threads `pMAX` uses the *Commander* and *Totalizer* encodings for the lower bound search and *Sorters* and *PB* encodings for the upper bound search. Although the ladder encoding performed better for the *at-most-one* constraint, it performed better mainly in the `bcp-mtg` and `pbo-mqc` benchmarks. However, the performance of the upper bound search in those benchmarks is much better than the lower bound search. Therefore, the main gains of the ladder encoding are already covered by the *at-most-k* encodings for the upper bound search. A similar reasoning applies to the PB representation for the upper bound search. Even though this representation is less effective in general, it is the best performing encoding for the `pbo-mqc` benchmarks. Hence, this portfolio of cardinality encodings allows for a large diversification of the search in all benchmarks.

With 8 threads `pMAX` can use more encodings and therefore can further increase the diversification of the search. For the upper bound search, all the four available encodings are used. For the lower bound search, we have selected the following encodings: *Commander*, *Totalizer*, *Ladder* and *Product*. The ladder encoding was now selected due to its overall robustness. On the other hand, even though the product encoding is less effective than other encodings, we have noticed that when it solves the instance, it can be faster than when using other encodings. This has already been observed before [11]. Hence, for speedup reasons, we have decided to include the product encoding on our portfolio of cardinality encodings for 8 threads.

Fig. 3. Cactus plot with running times of solvers



B. Solver Evaluation

We now compare our new parallel MaxSAT solver, pMAX, with other state-of-the-art sequential and parallel MaxSAT solvers. Figure 3 shows a cactus plot with running times of sequential and parallel solvers. The sequential solvers considered were QMaxSAT [4] (ranked 1st in the MaxSAT Evaluation 2010), pm2 [1] (ranked 2nd) and wbo [19], [20] (ranked 3rd). The parallel solvers considered were pwbo T4-CNF [22] and our new solver pMAX T4. SAT4J MAXSAT [18] and SAT4J MAXSAT RES//CP were not evaluated since their performance is not comparable to the remaining state-of-the-art partial MaxSAT solvers. For the 497 instances tested, SAT4J MAXSAT 2.2.3 and SAT4J MAXSAT RES//CP can only solve 277 and 290 instances, respectively.

The results are clear: when considering wall clock time, pMAX with 4 threads outperforms the best sequential and parallel solvers. When evaluating a parallel solver, the wall clock time is always considered since it measures the real time that a solver used to solve the instances. From a user point of view, real time is clearly more important than CPU time. On the other hand, if we analyze the CPU resources then parallel solvers with 4 threads are allowed to use four times more CPU time than sequential solvers. Figure 3 shows that pMAX T4 can outperform all sequential solvers even with a time limit of 1800 CPU seconds, i.e. within 450 wall clock seconds. Indeed, pMAX 4T is able to solve 389 instances in the same CPU time that QMaxSAT uses to solve 372 instances. To further analyze the resources point of view we increase the timeout of sequential solvers to 7,200 CPU seconds. With this experiment we are allowing the same CPU time for all solvers. Table III shows the number of instances solved with this new timeout. All sequential solvers can now solve more instances. In fact, PM2 is now able to solve more instances than QMaxSAT. However, pMAX 4T still outperforms all solvers.

In section IV we have described the sharing mechanism of pMAX. It is expected that sharing learned clauses can help to further prune the search space and boost the performance of the parallel solver. Figure 4 provides a scatter plot with the runtimes of pMAX 4T with and without sharing. Each point in the plot corresponds to a problem instance, where the x-axis corresponds to the runtime required by pMAX 4T with sharing and the y-axis corresponds to the runtime required by pMAX 4T without sharing. Instances that are trivially solved by both approaches (in less than 10 seconds) are not shown in the plot. Even though we can only solve a few more instances with sharing than without sharing, figure 4 clearly shows that sharing learned clauses speeds up the solver.

[4] http://www.maxsat.udl.cat/10/solvers/QMaxSat.pdf

TABLE III
NUMBER OF INSTANCES SOLVED WITH TIMEOUT 7,200S (CPU TIME)

# I	PM2	QMaxSAT	wbo	pwbo 4T-CNF	pMAX 4T
497	388	381	329	390	400

Fig. 4. Runtimes for pMAX 4T with and without sharing

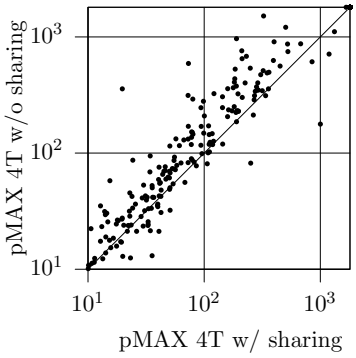


Fig. 5. Runtimes for pMAX 4T and pMAX 8T

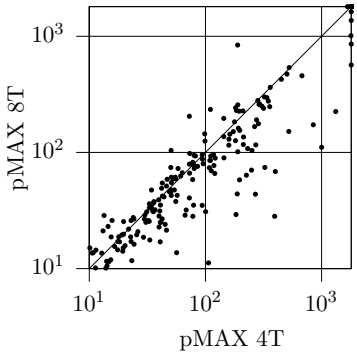


Figure 5 compares pMAX with 4 and 8 threads. pMAX 8T is able to solve more instances and to outperform pMAX 4T on most instances. This shows that even with 8 threads we are still able to increase the diversification of the search by adding different cardinality encodings.

Table IV shows an overview of the speedup on instances that were solved by wbo and the parallel solvers. pMAX 4T is almost 6× faster than wbo. This is due to the portfolio of cardinality encodings and to the fact that we are simultaneously searching on the lower and upper bound sides of the optimal solution. Moreover, pMAX 4T is 1.6× faster than our previous solver pwbo 4T-CNF. When using 8 threads, pMAX 8T is 8.5× faster than wbo and 1.4× faster than pMAX 4T.

TABLE IV
SPEEDUP ON THE 329 INSTANCES SOLVED BY ALL OUR SOLVERS

Solver	Time (s)	Speedup
wbo	67,947.41	1.00
pwbo 4T-CNF	18,015.69	3.77
pMAX 4T	11,382.91	5.97
pMAX 8T	7,990.10	8.50

VII. CONCLUSIONS AND FUTURE WORK

Several cardinality encodings have been proposed in the last decade. This paper examined a large number of cardinality encodings and evaluated their performance for solving the MaxSAT problem. Overall, the ladder encoding showed the best performance for the *at-most-one* cardinality constraints. As expected, when the number of variables is small it is better to use the pairwise encoding or PB representation. On the other hand, when the number of variables in the cardinality constraint is large, it is better to encode the *at-most-one* cardinality constraint into CNF. For the *at-most-k* cardinality constraint the sorter encoding showed the best performance. In general, it is better to translate the *at-most-k* cardinality constraint into CNF. However, when given a cardinality constraint of size n with k close to $n/2$, using a PB representation can be more effective than encoding the cardinality constraint into CNF. As future work, we propose to build a dynamic heuristic that given a portfolio of cardinality encodings and a cardinality constraint, chooses the more adequate encoding for that constraint. Such an heuristic can also be helpful for sequential solvers that use cardinality encodings.

Motivated with the existing diversity of effective cardinality encodings, we propose `pMAX`, a parallel partial MaxSAT solver that uses a portfolio of cardinality encodings and shares learned clauses between the different threads. `pMAX` clearly outperforms state-of-the-art sequential and parallel partial MaxSAT solvers on industrial benchmarks. Moreover, `pMAX 4T` is able to solve more instances than the best sequential solver with the same CPU time, i.e. using four times less wall clock time. Experimental results show that clause sharing has a strong impact on the solving speed. When using 8 threads, `pMAX 8T` scales well since it is 1.4× faster than `pMAX 4T`. Future directions include the implementation of other *at-most-k* encodings, namely, cardinality networks [3] and pairwise cardinality networks [9]. These encodings are also based on sorters and are expected to further improve our portfolio of algorithms.

ACKNOWLEDGMENT

This work was partially supported by FCT under research projects BSOLO (PTDC/EIA/76572/2006) and iExplain (PTDC/EIA-CCO/102077/2008), and INESC-ID multi-annual funding through the PIDDAC program funds.

REFERENCES

[1] C. Ansótegui, M. Bonet, and J. Levy, "Solving (Weighted) Partial MaxSAT through Satisfiability Testing," in *International Conference on Theory and Applications of Satisfiability Testing*, 2009, pp. 427–440.

[2] C. Ansótegui and F. Manyà, "Mapping problems with finite-domain variables into problems with boolean variables," in *International Conference on Theory and Applications of Satisfiability Testing*, 2004, pp. 1–15.

[3] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "Cardinality Networks: a theoretical and empirical study," *Constraints*, vol. 16, no. 2, pp. 195–221, 2011.

[4] G. Audemard and L. Simon, "Predicting Learnt Clauses Quality in Modern SAT Solvers," in *International Joint Conference on Artificial Intelligence*, 2009, pp. 399–404.

[5] O. Bailleux and Y. Boufkhad, "Efficient CNF Encoding of Boolean Cardinality Constraints," in *International Conference on Principles and Practice of Constraint Programming*, 2003, pp. 108–122.

[6] A. Biere, "Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010," *SAT Race, Solver Description*, 2010.

[7] M. Büttner and J. Rintanen, "Satisfiability Planning with Constraints on the Number of Actions," in *International Conference on Automated Planning and Scheduling*, 2005, pp. 292–299.

[8] J. Chen, "A New SAT Encoding of the At-Most-One Constraint," in *International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2010.

[9] M. Codish and M. Zazon-Ivry, "Pairwise Cardinality Networks," in *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2010, pp. 154–172.

[10] N. Eén and N. Sörensson, "Translating pseudo-Boolean Constraints into SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–26, 2006.

[11] A. M. Frisch and P. A. Giannaros, "SAT Encodings for the At-Most-k Constraint," in *International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2010.

[12] A. M. Frisch, T. J. Peugniez, A. J. Doggett, and P. Nightingale, "Solving Non-Boolean Satisfiability Problems with Stochastic Local Search: A Comparison of Encodings," *Journal of Automated Reasoning*, vol. 35, no. 1-3, pp. 143–179, 2005.

[13] Z. Fu and S. Malik, "On solving the partial MAX-SAT problem," in *International Conference on Theory and Applications of Satisfiability Testing*, 2006, pp. 252–265.

[14] I. P. Gent and P. Nightingale, "A new encoding of All Different into SAT," in *International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2004.

[15] Y. Hamadi, S. Jabbour, and L. Sais, "Control-Based Clause Sharing in Parallel SAT Solving," in *International Joint Conference on Artificial Intelligence*, 2009, pp. 499–504.

[16] ——, "ManySAT: a Parallel SAT Solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2009.

[17] W. Klieber and G. Kwon, "Efficient CNF Encoding for Selecting 1 from N Objects," in *International Workshop on Constraints in Formal Verification*, 2007.

[18] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2 system description," *Journal on Satisfiability Boolean Modeling and Computation*, vol. 7, pp. 59–64, 2010.

[19] V. Manquinho, J. Marques-Silva, and J. Planes, "Algorithms for Weighted Boolean Optimization," in *International Conference on Theory and Applications of Satisfiability Testing*, 2009, pp. 495–508.

[20] V. Manquinho, R. Martins, and I. Lynce, "Improving Unsatisfiability-Based Algorithms for Boolean Optimization," in *International Conference on Theory and Applications of Satisfiability Testing*, 2010, pp. 181–193.

[21] J. Marques-Silva and K. Sakallah, "GRASP: A new search algorithm for satisfiability," in *International Conference on Computer-Aided Design*, 1996, pp. 220–227.

[22] R. Martins, V. Manquinho, and I. Lynce, "Parallel Search for Boolean Optimization," in *RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 2011.

[23] S. Prestwich, "Variable dependency in local search: prevention is better than cure," in *International Conference on Theory and Applications of Satisfiability Testing*, 2007, pp. 107–120.

[24] C. Sinz, "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints," in *International Conference on Principles and Practice of Constraint Programming*, 2005, pp. 827–831.

[25] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, "Efficient conflict driven learning in boolean satisfiability solver," in *International Conference on Computer-Aided Design*, 2001, pp. 279–285.