Spring 4-20-2012

Native Cardinality Constraints: More Expressive, More Efficient Constraints

Jordyn C. Maglalang
jmaglala@iwu.edu

Native Cardinality Constraints: More Expressive, More Efficient Constraints

Jordyn C. Maglalang

Illinois Wesleyan University, Bloomington IL 61701, USA
jmaglala@iwu.edu

Abstract. Boolean cardinality constraints are commonly translated (encoded) into Boolean CNF, a standard form for Boolean satisfiability problems, which can be solved using a standard SAT solving program. However, cardinality constraints are a simple generalization of clauses, and the complexity entailed by encoding them into CNF can be avoided by reasoning about cardinality constraints *natively* within a SAT solver. In this work, we compare the performance of two forms of native cardinality constraints against some of the best performing encodings from the literature. We designed a number of experiments, modeling the general use of cardinality constraints including crafted, random and application problems, to be run in parallel on a cluster of computers. Results show that native implementations substantially outperform CNF encodings on instances composed entirely of cardinality constraints, and instances that are mostly clauses with few cardinality constraints exhibit mixed results warranting further study.

1 Introduction

Boolean Satisfiability (SAT) is the problem of finding an assignment to a set of Boolean variables, which can be set to either True or False, in a Boolean formula such that the entire formula evaluates to True. This definition makes SAT a prime example of a decision problem, which is simply a problem resulting in a "yes-or-no" response. The value of SAT is observed in how other decision problems can be encoded into SAT. In this event, a Boolean formula is generated and its satisfiability will have a direct meaning for the original problem. Though decision problems are often intractably large and generally difficult to find solutions for in a reasonable time, there exist programs called "SAT Solvers" that, in practice, are able to solve a large number of SAT problems.

With regards to Boolean formulas, cardinality constraints are restrictions on the number of variables within a given set that can be assigned True. These constraints show up in a number of different real-world problems including course scheduling, formal hardware verification, radio frequency assignment [7,4,5], etc.

Such a constraint can be considered a decision question where the constraint is satisfiable so long as number of variables within the constraint is equal to, above, or below the defined bound, depending on the specific implementation. The general approach to using cardinality constraints is to encode them into

Boolean formulas and pass the newly encoded formula into an unmodified, "black box," SAT solver. These solvers accept the most common normal form SAT, conjunctive normal form (CNF). This allows for anyone interested in cardinality constraints to find solutions without having to have any expertise in how SAT solvers function.

Though semantically simple cardinality constraints are translated, or encoded, into SAT which significantly increase the time and space complexity of the original formula, which can be avoided by reasoning about them *natively* within the solver. Augmenting a solver with native cardinality constraints involves creating separate data structures to store cardinality constraints as well as a separate code path for handling them. A solver with this ability is able to add a single native cardinality constraint instead of numerous clauses and/or auxiliary variables avoiding the additional space and complexity incurred from CNF encodings. An unoptimized native implementation of cardinality constraints was included in earlier versions[1] of MiniSAT [8], a modern SAT solver, for the purpose of displaying the solver's ability to be easily extended. A recent evaluation of cardinality constraint encodings [2] made use of an "SMT" solver to reason about cardinality constraints in place of encoding them into SAT. This "coupling" of two solving engines does not permit the tight integration of cardinality into the SAT solver done in this work, and it did not perform well compared to CNF encodings. Marques-Silva and Lynce [13] explored modifications to a SAT solver that improved its efficiency when using a particular CNF encoding, but it still faced the inherent space complexity of such encodings and was limited to AtMost constraints with a bound of 1.

The aim of this work is to evaluate the performance of a solver handling cardinality constraints with both Native and CNF encodings. Section 2 formally defines SAT, CNF, cardinality constraints, and the relationship between clauses and cardinality, and Section 3 describes the CNF encodings used during the evaluation. Section 4 discusses the implementation of native cardinality constraints within a SAT solver, followed by an experimental evaluation which is laid out in Section 5 and discussed in Section 6.

2 Preliminaries

The Boolean Satisfiability (SAT) problem involves determining if a satisfying assignment to a Boolean formula exists and finding that assignment if possible. Boolean formulas consist of literals, variables or their negation which can be set to either True of False, that are in conjunction or disjunction with each other, where a conjunction is the logical AND and a disjunction is the logical OR. A Boolean formula is then satisfiable if there exists an assignment to all of its literals such that the formula evaluates to True, otherwise, it's considered unsatisfiable.

[1] MiniSAT v1.12b is the last version including the Constraint class and its example subclass AtMost.

Though Boolean formulas can exist as any set of literals together with logical operators, a number of normal forms have been defined in order to standardize the problem. The most common normal form, Conjunctive Normal Form (CNF), is defined as the conjunction of clauses C_i, where each clause is a disjunction of literals.

$$\varphi = \bigwedge_{i=1\ldots m} C_i \qquad C_i = \bigvee_{j=1\ldots k_i} a_{ij}$$

For example, for the following CNF formula: $(a \vee b) \wedge (\neg b \vee c \vee \neg d) \wedge (\neg a \vee \neg c)$

- $(\neg b \vee c \vee \neg d)$ is a clause
- The following is a satisfying assignment:
  - $a \leftarrow$ True
  - $b \leftarrow$ False
  - $c \leftarrow$ False
  - $d \leftarrow$ True

There exists a class of algorithms called SAT-Solvers that determine whether a particular CNF formula is satisfiable. One category of such solvers are the conflict-driver clause-leaning (CDCL) solvers [8,15,16] which make use of various techniques for reducing the overall complexity of solving a CNF formula. MiniSAT [8], the solver we used in this study, is a state-of-the-art CDCL solver with simple and efficient code.

A *cardinality constraint* places a bound on the number of literals within a given set that can be assigned True. Given a set of n literals $\{a_1, a_2, \ldots, a_n\}$ and an integer bound k, s.t. $0 \leq k \leq n$, a cardinality constraint is defined as:

$$\sum_{i=1}^{n} a_i \gtreqless k$$

Here, $\gtreqless$ is any relation from the set $\{\leq, =, \geq\}$, forming *AtMost*, *Equals*, and *AtLeast* constraints, respectively. Note that since a clause is the disjunction of literals only a single literal must be assigned true for the clause to be true making it equivalent to an AtLeast constraint with a bound of 1:

$$(a_1 \vee a_2 \vee \ldots \vee a_n) \equiv \text{AtLeast}(1, \{a_1, a_2, \ldots, a_n\})$$

For example, the clause $(\neg b \vee c \vee \neg d)$ would become the following cardinality constraint:

$$\text{AtLeast}(1, \{\neg b, c, \neg d\})$$

Because a clause can be expressed as a cardinality constraint, these constraints actually subsume — and are more expressive than — clauses.

It should be noted, however, that any cardinality constraint can be expressed with AtMost constraints:

$$\text{AtLeast}(k, \{a_1, \ldots, a_n\}) \equiv \text{AtMost}(n-k, \{\neg a_1, \ldots, \neg a_n\})$$
$$\text{Equals}(k, \{a_1, \ldots, a_n\}) \equiv \text{AtLeast}(k, \{a_1, \ldots, a_n\}) \wedge \text{AtMost}(k, \{a_1, \ldots, a_n\})$$

Therefore, in order to represent any type of cardinality constraint, only an implementation of either AtMost or AtLeast constraints is necessary.

For example, the clause $(\neg b \vee c \vee \neg d)$ can also be define as:

$$\text{AtMost}(2, \{b, \neg c, d\})$$

3 Encodings

Substantial research has been devoted to finding and improving upon encodings of cardinality constraints into CNF. Here, we describe some of the most recent work and experimental results regarding the CNF encodings used in the comparison in Section 6.

One compact encoding of cardinality constraints is derived from encoding each constraint as a Binary Decision Diagram (BDD) [9]. The BDD form of an AtMost constraint is translated into CNF by modeling each node of the BDD as a multiplexer or if-then-else (ITE) operator, each of which can be encoded in three clauses. This encoding requires $(k+1)(n-k)$ multiplexers, hence $\mathrm{O}(n \cdot k)$ clauses, and it preserves arc-consistency, as proven in [9], which means that as the solver makes decisions the constraint will correctly, and immediately, assign the remainder of its literals if its bound is reached.

Another CNF encoding, adopted by Eén and Sörennson [9], is built from odd-even sorting networks due to Batcher [3]. A network of comparators sorts the values of an AtMost's literals, and setting $n - k$ of the outputs to False enforces the bound. Asín, et al. [2] develop the concept further, introducing half sorting and half cardinality networks. They observe that the comparators used to build these networks only require 3, not 6 clauses apiece, reducing the size of the encoding (though still $\mathrm{O}(n \log^2 k)$ clauses). The half sorting/cardinality networks provide arc consistency as well as incremental strengthening, which means that a tighter constraint can be obtained from any particular constraint without any new variables or clauses, simply by setting a single variable to False. In their evaluation, Asín, et al. found that their cardinality networks tended to out-perform all other encodings tested, especially for large constraints with a small bound. For instances in the "Tomography" family, however, they found that the BDD encoding performed significantly better.

Parberry [17] proposes an alternative to Batcher's sorting networks, which he calls Pairwise Sorting Networks, using an alternative splitting method when forming the sorting network. Codish and Zazon-Ivry [6] apply these sorting networks to cardinality constraints, showing that they form smaller networks than odd-even networks and have "significantly better propagation properties." The "half comparator" introduced by Asín, et al. is applicable to pairwise cardinality networks as well. Evaluating the encoding on Boolean cardinality matrix problems, they find that the size of the network significantly affects the performance of a solver, and their results indicate a runtime advantage for pairwise networks.

4 Implementation

In this work, we extend a modern SAT solver to include the ability to handle cardinality constraints natively alongside standard CNF clauses. MiniSAT, the solver on which our implementation is based, accepts DIMACS CNF which is a standard format for specifying CNF instances. We extend this CNF format (See section 4.2) to allow for cardinality constraints to be specified alongside clauses.

In MiniSAT, the solver interacts with clauses primarily in two procedures, Propagate and Analyze.

Propagate is the mechanism by which the solver informs a clause of a new assignment and finds whether that assignment induces further assignments or causes a conflict. In clauses, further assignments are induced when a clause becomes *unit*, meaning all but one of its literals has been assigned False and the remaining literal must now be assigned True. In an AtMost constraint, propagation occurs when the number of literals assigned True reaches the bound, at which point the remaining, unassigned literals must be assigned False.

Analyze is called when a conflict is found to determine the reason for the conflict. It will produce a new, "learned" clause that will be added to the overall formula which prevents the current assignment from happening again. This is also known as a conflict clause or a blocking clause. This method requires, from any constraint it inspects, a list of all earlier assignments that caused the constraint to propagate some given assignment(s). A clause can satisfy that requirement by listing all of its literals which were assigned False while an AtMost constraint can satisfy that requirement by listing all of its literals that have been assigned True.

We implement AtMost constraints in two different ways. The first follows the AtMost code included in MiniSAT v1.12, keeping a simple counter within each AtMost to determine when it has reached its bound.

The second implementation, provided by Mark Liffiton, generalizes the common strategy for determining when a clause propagates a new assignment. In this "two-watched-literal" strategy only two literals within a clause are watched at any given time. If one of these literals is assigned, a new literal in the clause is chosen to be watched. If there are no such literals then the final literal being watched is assigned to True and is propagated. In our "m-watched-literal" strategy an AtMost constraint must propagate the negation of all remaining unassigned literals when its count of True literals grows from $k-1$ to k. In this case, $n-k$ literals remain that must be assigned False for the constraint to remain satisfied, and so $m = n-k+1$ literals are watched to detect the condition in which propagation must occur.

4.1 Counter-Based Implementation

The counter-based implementation follows that in MiniSAT 1.12b and directly implements the semantics of an AtMost constraint. It maintains a count of how many of its literals have been assigned True and propagates the negation of the remainder once its bound is reached.

```
Propagate_AtMost(constr, decisionLevel)
 1. constr.incrementCounter(decisionLevel)
 2. if constr.boundExceeded then
 3.   return conflict
 4. else if constr.boundReached then
 5.   unassigned ← {}        ◁ container for unassigned literals
 6.   if constr.getUnassigned(unassigned) then
 7.     return conflict
 8.   else
 9.     for lit in unassigned do
10.       enqueue(¬lit)
```

Fig. 1. The Propagate algorithm for a counter-based AtMost constraint.

As in MiniSAT 1.12, AtMost constraints are stored separately from the clauses in this implementation, with a separate object for each. A dummy clause is added to the solver for each AtMost created, containing a flag indicating it is not a real clause and a pointer to that AtMost's object. Any procedure handling a clause checks the flag and executes a separate code path for an AtMost constraint, de-referencing the pointer to access the constraint's data as needed.

Pseudocode for propagating a new assignment into a counter-based AtMost constraint `constr` is shown in Figure 1. The function increments the constraint's counter, signaling a conflict if its bound is exceeded. We include an optimization in the getUnassigned() function (see figure 2) which helps detect conflicts earlier. This optimization arises from the fact that while multiple assignments can be enqueued at a time only one can be propagated at a time. When a constraint is being checked, if its bound is reached, due to the currently propagated literal, then the getUnassigned() function will collect the unassigned literals, counting all those that are assigned as it does this. If the number of assigned literals are greater than the bound then a conflict is signaled, otherwise Propagate() will simply enqueue the negation of the remaining unassigned literals.

One main function that cardinality constraints must support are unassignments, or "backtracking", which occur throughout solving. In order to support this, a cardinality constraint is required to wind back its counter to whatever it was when the solver was previously at its current "decision level" (after backtracking). The decision level is simply the number of decisions the solver has made, representing its current depth in the search tree. To allow for backtracking, each constraint stores a history of its counter as a stack of (`decisionLevel`,`count`) pairs, recording a new pair whenever one of its literals is assigned True and "rewinding" the count based on the new decision level when the solver backtracks. The pseudocode in Figure 3 illustrates how an AtMost's counter is updated during propagation and backtracking. The requirements of Analyze are met with a simple method (not shown) that scans the AtMost's literals and returns those that are assigned True.

```
getUnassigned(out)
1. n ← 0        ◁ counter for assigned literals
2. for lit in constr do
3.    if solver.value(lit) = Undefined then
4.       out.push(lit)
5.    else if solver.value(lit) = True then
6.       n ← n + 1
7. return n > bound
```

Fig. 2. The getUnassigned algorithm for a counter-based cardinality constraint.

```
incrementCounter(level)
1. counter ← counter + 1
2. if level = history.last.level then
3.    history.last.count ← counter
4. else
5.    history.push({counter,level})
```

```
cancelUntil(level)
1. while (history.last.level > level)
2.    history.pop
3. counter ← history.last.counter
```

Fig. 3. Maintaining an AtMost constraint's counter

This implementation is simple, but it has two drawbacks relative to a watched-literal implementation. First, it requires a watch on every one of its literals to maintain the correct count. Second, the count must also be updated when the constraint's literals are *un*assigned, requiring additional work when the solver is backtracking that is not required in a watched-literal implementation. With these facts in mind, we expect that the watcher-based implementation should outperform this counter-based implementation.

4.2 CNF+

The standard format for problems in SAT, Conjunctive Normal Form, is generally stored in the DIMACS format composed of a header, containing any comments and a formula description, followed by a list of clauses. The clauses are separated by "0" and contain an integer representing the variable number and sign. *CNF+*, the format we defined for our solver, is an extension of this format and uses a similar convention for specifying constraints. Clauses in this format are defined in the same way they were in DIMACS, while constraints are defined with a list of integers, representing literals and their signs, followed by a relational operator and a bound. This new format is defined by the Backus-Naur Form in Figure 4.

⟨formula⟩ ::= ⟨sequence of comments⟩,⟨description⟩,⟨sequence of clauses or constraints⟩
⟨sequence of comments⟩ ::= ⟨comment⟩ [⟨sequence of comments⟩]
⟨comment⟩ ::= “c” ⟨any sequence of characters other than EOL⟩ EOL
⟨description⟩ ::= “p” ⟨one space⟩ “cnf+” ⟨num var⟩ ⟨num clause⟩ ⟨num constraint⟩
⟨num var⟩ ::= ⟨unsigned integer⟩
⟨num clause⟩ ::= ⟨unsigned integer⟩
⟨num constraint⟩ ::= ⟨unsigned integer⟩
⟨sequence of comments or constraints⟩ ::= ⟨clause⟩ | ⟨constraint⟩ [⟨sequence of clauses or constraints⟩]
⟨clause⟩ ::= ⟨sequence of terms⟩ “0”
⟨constraint⟩ ::= ⟨sequence of terms⟩ ⟨relational operator⟩ ⟨one space⟩ ⟨unsigned integer⟩ “0”
⟨sequence of terms⟩ ::= ⟨term⟩ [⟨sequence of terms⟩]
⟨one space⟩ ::= “ ”
⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨digit⟩ ⟨unsigned integer⟩
⟨integer⟩ ::= ⟨unsigned integer⟩ | [“-”] ⟨unsigned integer⟩
⟨relational operator⟩ ::= ≤|≥| =
⟨term⟩ ::= ⟨integer⟩ ” ”

Fig. 4. Backus-Naur Form definition for CNF+ file format

5 Methodology

The aim of this paper is to show how a native approach to handling cardinality constraints compares to using various CNF encodings. Since CNF encodings all require the addition of extra clauses and/or auxiliary variables, the amount of space required to store the problem, along with the information learned, can strain solving. We are therefore interested in evaluating both the time and space complexity of each approach.

We modeled our experiments after the SAT* Competitions, which pit the best SAT solvers against each other, dividing the evaluations into three parts: crafted, application and random. Due to the nature of the problem types, each part offers a different insight into the performance of cardinality constraints, allowing for a broader, more general analysis of cardinality constraints.

With no existing library of cardinality constraint benchmarks, we were required to generate the majority of ours, constructing a CNF+ generator[2] for most problem types. The exceptions to this are the MSU4 benchmarks which we received from Asiń, et al. and the benchmarks that we used for CAMUS, which we acquired from the industrial track of the SAT 2011 competition website [1].

5.1 Crafted

Crafted instances offer various forms of cardinality constraints. Typically these problems are described in terms of the form their constraints take, either *AtMost*

[2] Our generators are available at `https://jmaglala@github.com/jmaglala/CNFP-Generators.git`.

1 or *AtMost* k. Being a fairly common form, *AtMost* 1 constraints have been the sole focus of some comparative studies [13].

For our crafted problems we used the n-queens, tomography and Word design for DNA computing on surfaces problems.

***n*-queens** The n-queens problem is the problem of finding an arrangement for n queens on an $n \times n$ chess board so that no two queens can attack each other.

In order to define an instance in terms of cardinality constraints we first assign a Boolean variable for each position in an $n \times n$ grid representing the chess board. A queen present at any given location is then represented by setting the corresponding variable to True. From this, the cardinality constraints required follow directly from the problem definition, all of which will be *Equals* 1 (though recall that an *Equals* constraint can be represented by 2 *AtMost* constraints). In total, $6n - 6$ constraints will be generated: $2n$ constraints to restrict the number of queens on each row and column and $4n - 6$ constrains to restrict the number of queens in each diagonal.

Tomography The tomography problem [11] is the problem of determining which cells of an $n \times n$ grid are filled if all that is known is the grid size n and the number of cells filled in each row, column, and diagonal.

Similarly to the n-queens problem, in order to define an instance in terms of cardinality constraints we construct an $n \times n$ grid of Boolean variables where a True assignment represents a filled cell. For any given n we can generate a number of different instances by randomly filling cells within the grid. Tomography instances will contain a total of $6n - 6$ *Equals* k constraints, again with $2n$ constraints to represent row and column and $4n - 6$ constrains to represent each diagonal.

Word Design for DNA Computing on Surfaces Word design for DNA computing on surfaces that arises out of bioinformatics and is an example of a general constraint satisfaction problem that can be expressed with cardinality constraints. It is initially described in[10] and is featured on CSPLib [3]. The problem is to find as large a set of strings of length 8 (words) as possible with the following conditions:

1. Each word has 4 symbols from { C,G }
2. Each pair of distinct words differs in at least 4 positions
3. Each pair of words x and y (where x and y may be identical) are such that x^R and y^C differ in at least 4 positions. Here, $(x_1, \ldots, x_8)^R = (x_8, \ldots, x_1)$ and $(y_1, \ldots, y_8)^C$ is the Watson-Crick complement of $(y_1, \ldots, y_8)$, i.e the word where each A is replaced by a T, and vice versa, and each C is replaced by a G, and vice versa.

[3] http://www.cs.st-andrews.ac.uk/~ianm/CSPLib/prob/prob033/index.html

Word design for DNA computing on surfaces is considered to be a combinatorial optimization problem which is not a decision problem. Recall that decision problems must ask "yes-or-no" questions, here the problem is asking instead for the *largest possible* solution. Optimization problems can, however, be solved with a series of decision problems in an incremental fashion. The decision problem for Word design becomes thus: does there exist a set of size n for which the given conditions hold.

Representing this problem in terms of cardinality constraints is non-trivial since its conditions deal with restrictions on pairs of words. A word is defined by a set of 36 Boolean variables, that present a "one-hot" encoding for each of the 8 letters in the word. In this case, a one-hot encoding means that for each position in the word we use 4 variables where only a single one may be assigned True, representing the 4 possible letters [4].

For the first condition, a single cardinality constraint is generated for each word, making an *Equals* 4, where the variables in the constraint are all those representing a 'C' or 'G' at each possible position. For the second condition, we create additional variables to represent a comparison between the individual positions of two words. So, for words x and y, a list of Boolean variables z such that z_n is True if $x_n = y_n$. For each pair of words we then generate cardinality constraints restricting *AtMost* 4 of these positions to be the same (which is equivalent to *AtLeast* 4 are different). The final condition is represented in a similar fashion, but for the words x^R and y^C, so z_n would be true if $x_n^R = y_n^C$.

5.2 Application

Application problems are those which arise out of industrial problems and give good insight into how cardinality constraints are used in practice. Cardinality constraints have been applied in several algorithms that analyze unsatisfiable SAT instances, including CAMUS [12] and the MSU* suite of maximum satisfiability (Max-SAT) algorithms [14]. Each augments clauses from a CNF instance with new variables. These new "relaxation" variables are controlled by either algorithm and are able to effectively turn-off or remove clauses without actually removing them from the formula. They can then add one or more *AtMost* constraints over the relaxation variables to place bounds on the number of clauses that are "enabled" at any point. It is important to note that instances of these two problems will have few cardinality constraints and will be mostly comprised of CNF clauses. In the case of CAMUS there will only be a single cardinality constraint used per benchmark.

For our experiment, CAMUS was reimplemented using MiniSAT 2.2.0 and run against all benchmarks used in the SAT 2011 Competition MUS track [1] minus four that are satisfiable. The authors of [2] provided us with the set of about 14,000 MSU4 instances used in their evaluation of CNF encodings.

[4] Note that it would be possible to use only 2 variables per position with a binary encoding of the letters, however this would increase the complexity of the generator.

5.3 Random

While randomly generated instances may not accurately reflect how cardinality constraints are used in practice, they represent the most general application of cardinality constraints. The performance of a solver on randomly generated instances will be determined chiefly by the performance of the cardinality constraints themselves and will be less affected by the relationship between constraints.

There are multiple parameters to consider when generating random instances. Our generator takes: the maximum number of variables n in the formula, the ratio of variables to constraints r, and the size of each constraint k. Each instance thus contains $n \cdot r$ AtMost constraints, with bounds randomly selected between 1 and $k-1$, inclusive. Because variables are randomly selected while generating cardinality constraints it is very possible that a particular variable was not used. The entire formula must be regenerated until it can guarantee that all variables are used and while this is definitely feasible for small instances, it can potentially take an infinite amount of time. When looking at other work done in random CNF generation, like van Gelder's `mkcnf` [18], there is no such guarantee, and therefore our generator does not guarantee that all variables are used.

6 Results

All of the encodings were implemented on the "core" MiniSAT 2.2.0 solver, as opposed to the "simp" solver, which contains code for preprocessing, where an attempt is made to first simplify the input formula. This allows the experiments to focus on the performance of the constraints themselves, providing an "apples-to-apples" comparison with equivalent constraints given to each implementation.

The experiments were run on a Linux cluster with an AMD Phenom II 965 quad-core 3.4GHz processor and 8GB of RAM per node. Unless otherwise noted, every run was given a 600 second timeout and an 1,800 MB memory limit. The native m-watched-literal implementation is labeled "Native(w)" ('w' for **w**atched literals), while the **c**ounter-based implementation is labeled "Native(c)."

6.1 N-Queens

The runtimes for solving n-queens instances for n = multiples of ten from 10 up to 1800 are shown in Figure 5, and Figure 6 shows peak memory usage. The two native implementations substantially outperform all CNF encodings in both runtime and memory usage, scaling to $n = 1200$ and 1500, while the best CNF encoding only reaches n=430 before timing out. Furthermore, both native implementations use only half of the memory limit by $n = 1800$, while the most memory-efficient CNF encoding reaches the memory limit around $n = 500$.

Excluding BDD, all of the CNF encodings reach the memory limit much sooner than the Native encodings because of the memory required to store the

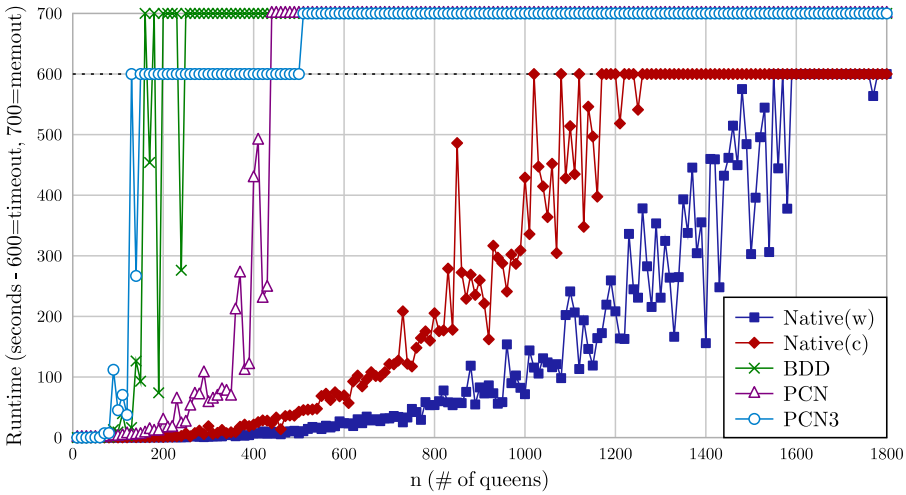

Fig. 5. n-Queens: Runtime for each constraint type.

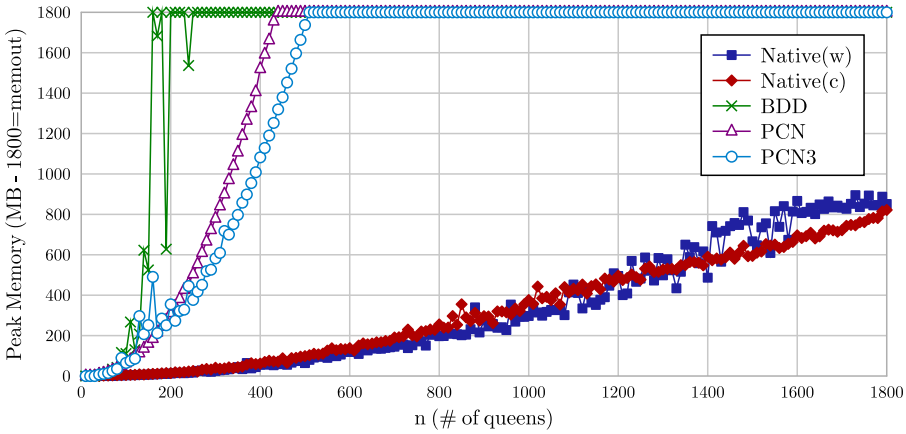

Fig. 6. n-Queens: Peak memory usage for each constraint type.

original formula. This is due to the sheer number of additional clauses and auxiliary variables that are added to the formula during the parsing of input. Interestingly, though the BDD encoding first exhausts the memory limit at $n = 180$ it is not a result of the size of the additional clauses and variables. For $180 \geq n \geq 300$, BDD is reaching the memory limit while solving though after 300 the size of the problem with additional clauses and variables exceeds the memory limit.

6.2 Tomography

We created tomography instances for $n = 20, 21, \ldots, 47$ which provide runtimes ranging from our measurement precision up to the timeout. For each size we generated 10 random instances, for a total of 280 instances. The runtimes for these instances are shown in a cactus plot[5] in Figure 7.

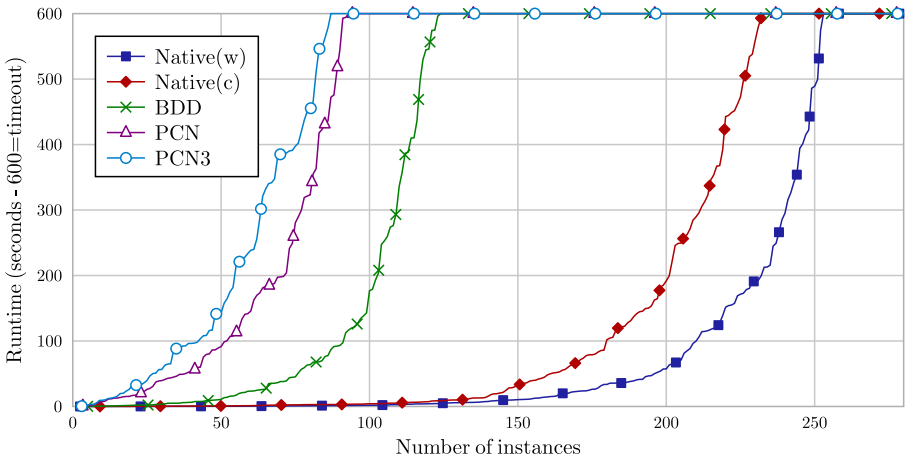

Fig. 7. Tomography: Cactus plot of runtimes for each constraint type.

Again, the native implementations substantially outperform all CNF encodings in both runtime and memory usage. Native(w) solved 252 instances within the timeout, while BDD, the best-performing CNF encoding, solved 123 of 280. The tomography results in [2] had the BDD encoding outperforming the others tested, including their native-like implementation, "SMT"; the results here show both our native implementations greatly outperforming the BDD encoding, illustrating the importance of the tight coupling of the constraints that is done in both Native(c) and Native(w).

[5] For this problem, the cactus plot first sorts instances by runtime, and it then plots them in increasing order, capturing the number of instances that are completed by a particular time but not allowing for pairwise comparisons between data sets.

6.3 Word Design

We generated instances of this problem for all values of n up to 85. Figure 8 shows the runtime for each type of constraint on each problem size. Here, Native(w) is able to complete one additional instance within the timeout (the complexity of the problem appears to rise very rapidly after $n = 80$) compared to PCN3 and BDD, and it requires significantly less time for all instances on which they finish within the timeout. In the memory usage plot, Figure 9, we see that the native implementations used much less memory than the CNF encodings until around $n = 80$, when Native(w) began requiring a great deal more memory. Note that after $n = 80$ all other constraint types time out, and the memory usage beyond $n = 80$ is in fact evidence that Native(w) is the only constraint type making significant search progress within the timeout.

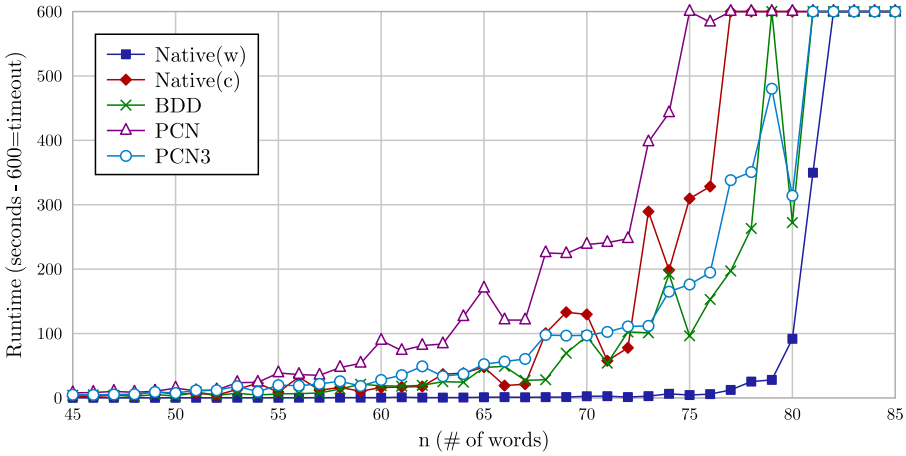

Fig. 8. Word Design: Runtime for each constraint type.

6.4 CAMUS/MSU4

The first phase of CAMUS is an anytime algorithm,[6] and its results can be used even if an instance times out. Therefore, we have plotted both its runtime and the rate at which it returned MCSes in Figure 10. On these instances, PCN3 outperformed PCN and BDD on nearly every benchmark, so we show only PCN3 and the two native implementations. The results show that Native(w) again outperforms Native(c), but its results versus PCN3 are mixed. Native(w) finishes before PCN3 in 30 instances, compared to 26 in which PCN3 completes

[6] An anytime algorithm can return a valid solution to a problem even if it is interrupted, i.e times out or runs out of memory.

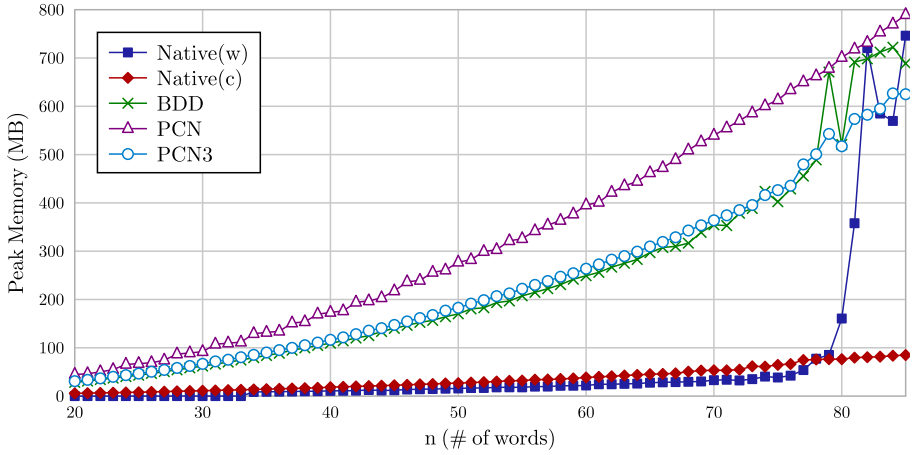

Fig. 9. Word Design: Peak memory usage for each constraint type.

more quickly. The rate of output can show a difference even when runs time out; PCN3 has a higher rate in 166 cases and a lower rate in 40.

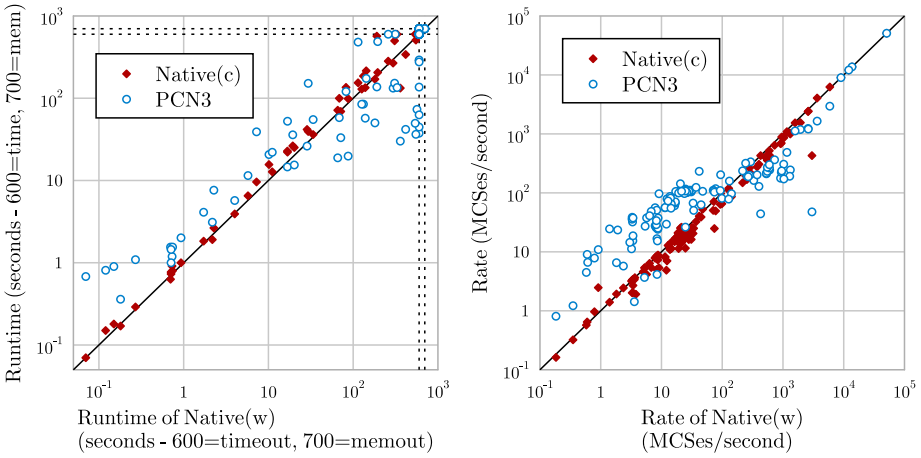

Fig. 10. CAMUS: Runtime and rate of output for Native(w) (x-axis) vs others.

For the MSU4 algorithm we randomly selected 10% of the instances that we received from the authors of [2] for our experiments. Figure 11 contains a logarithmic cactus plot illustrating the distribution of runtimes for each constraint type. Here, again, the results are mixed. PCN3 tends to outperform all other implementations, and it times out on 20 instances compared to 111 on which Na-

tive(w) reaches the timeout. Figure 12 directly compares Native(w) with PCN3; while PCN3 tends to perform better, the runtimes vary considerably between the two implementation, with Native(w) at times being orders of magnitude faster than PCN3.

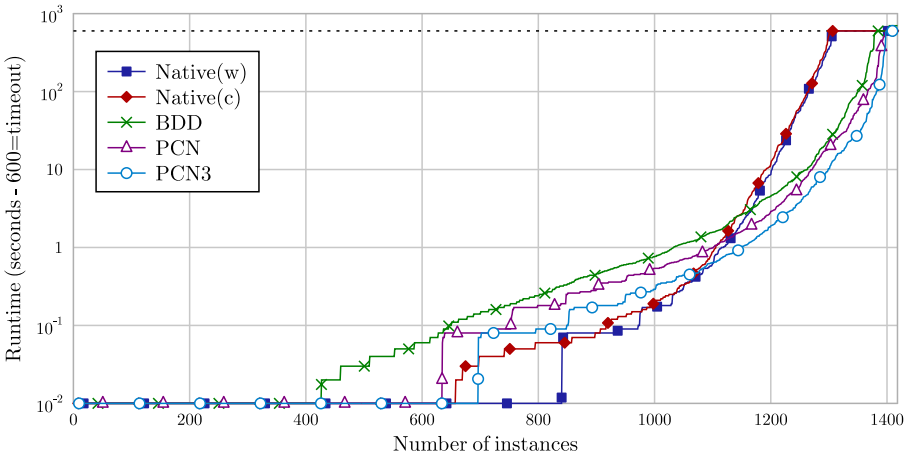

Fig. 11. MSU4: Logarithmic cactus plot of runtimes for each constraint type.

Recall that the AtMost constraints make up a very small part of each instance in these applications, and so the overall efficiency of the solver is still primarily driven by processing and managing regular clauses.

In these applications, the AtMost constraints control relaxation variables that enable and disable clauses, thus any bias an AtMost induces on the order of searching their assignments could have a very large effect on the size of the search tree.

We took a closer look at how PCN3 and Native(w), the best CNF and Native encodings for the application instances, and analyzed the differences in the size of the search tree alongside runtime. For these experiments we excluded all instances that were solved via unit propagation. [7] We exclude instances whose runtimes fall below a minimum threshold as their results are much less meaningful. Instances that are shown that completed in under 0.01s, which is the minimum accuracy of our experiments, are plotted as 0.005. While we are not able to accurately measure these instances' runtimes, they can still be used to produce valuable comparisons that would not be possible were their runtime left at 0s. In figure 13, the left plots the ratio of the number of decisions made by the Native(w) and PCN3 against the ratio of the runtime for Native(w) and PCN3.

[7] An instance is solved by unit propagation when a solution is found by propagating unit clauses, i.e clauses with only a single literal, that force the assignment of particular variables.

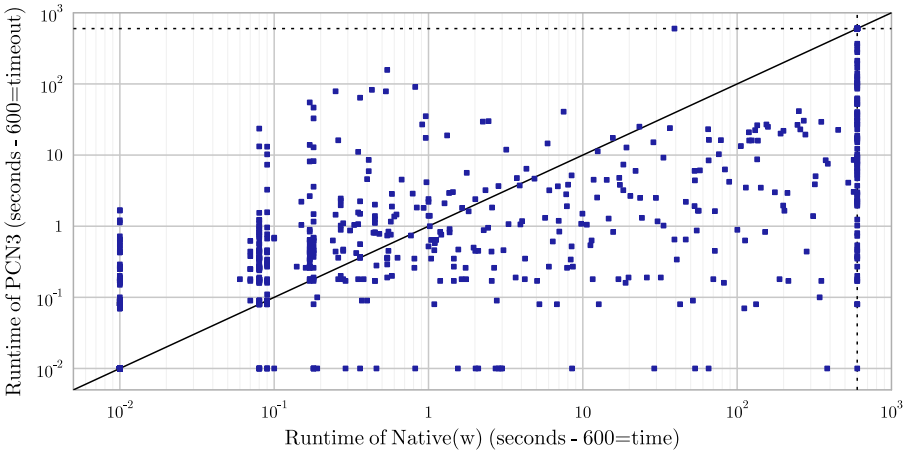

Fig. 12. MSU4: Runtime for Native(w) (x-axis) vs PCN3.

Here we can see that for the instances in which Native(w) is faster, those with a runtime ratio below 1, the ratio of decisions is about 1. For the instances in which PCN3 is faster the ratio of decisions begins to grow, i.e the size of Native(w)'s search tree is growing larger than PCN3's. The right of figure 13 plots the rate of decisions made for each instance. Here we can see that the distribution of instances is mostly below the diagonal, indicating that the decision rate was great for Native(w). Of the 709 instances, Native(w) had a faster rate than PCN3 in 522 whereas PCN3 was faster in 187.

We found that in 89% of the instances where Native(w) was slower, the number of decisions it made was larger than that of PCN3. In summary the difference in runtime observed in the MSU4 experiments between Native(w) and PCN3 seems to be largely due to the increase in the size of Native(w)'s search tree. PCN3 outperforms Native primarily by producing a smaller search tree. The remaining difference in runtime are not completely clear, and while the search tree does largely affect performance, the mechanism that increases its size is still unexplained.

6.5 Random

The final set of pure-cardinality instances are randomly generated. To produce an "interesting" set of benchmarks, we set $k = 10$ and searched for the phase transition, the ratio r at which instances are half satisfiable and half unsatisfiable. We found that this ratio varies with n, and with $n = 10^6$ (a size yielding useful runtimes), the phase transition is near $r = 0.20$. Therefore, we generated 100 instances each for $n = 10^6$, $k = 10$, and $r \in \{0.180, 0.184, 0.188, \ldots, 0.220\}$.

Figure 14 shows the runtimes for these instances in a log-log scatter plot. PCN has been omitted from the scatter plot for clarity; it reached the memory

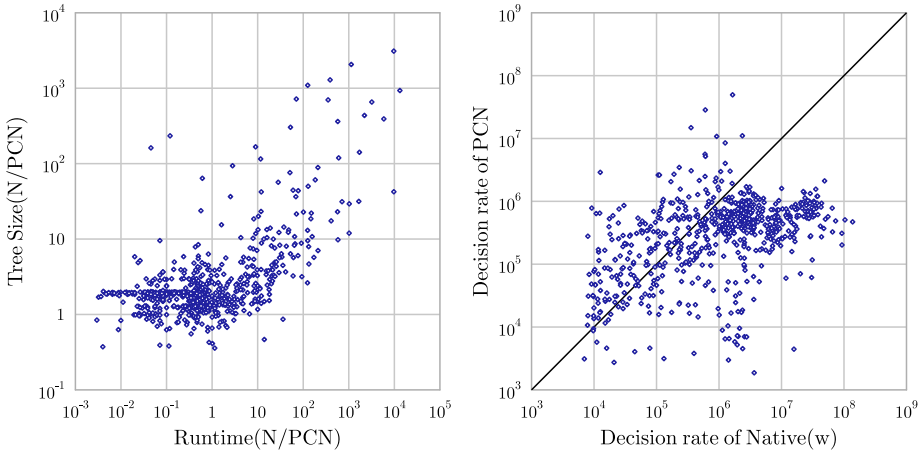

Fig. 13. Logarithmic scatter plots: (left) comparing the ratio of tree size between Native(w) and PCN3 with the ratio of their runtimes, (right) decision rate of Native(w) vs PCN3 on the MSU4 instances.

limit on all but one instance out of the 1100 tested. The figure shows a clear clustering of runtimes into two groups; the unsatisfiable instances, regardless of r, completed faster than those that were satisfiable for all constraint types. This is likely due to the fact that determining a formula is satisfiable requires fully exploring the search space to find a satisfying assignment, as opposed to simply finding a set of conflicts which prevent such an assignment. Native(w) again consistently outperforms Native(c), and all CNF encodings are at least an order of magnitude slower. Here, the BDD encoding edges out PCN3 to be the fastest CNF encoding.

7 Conclusion

We have demonstrated two native implementations of cardinality constraints within a state-of-the-art SAT solver producing an efficient "cardinality solver." Native implementations greatly outperform CNF encodings of cardinality constraints on all pure-cardinality instances tested, and instances with a mix of clauses and cardinality constraints exhibited mixed results. The watcher-based native implementation achieves the fastest decision rate; however, due in part to the increasing size of the search tree, PCN3 performs better in some cases. Given that cardinality constraints have an increased expressive power over CNF and that they are relatively simple to implement it is worthwhile to augment current SAT solvers to enable them to handle cardinality constraints.

A number of directions of future research arise from this work. Though we have looked at how cardinality implementations affect applications that are normally translated into CNF with encoded constraints, like CAMUS and MSU4,

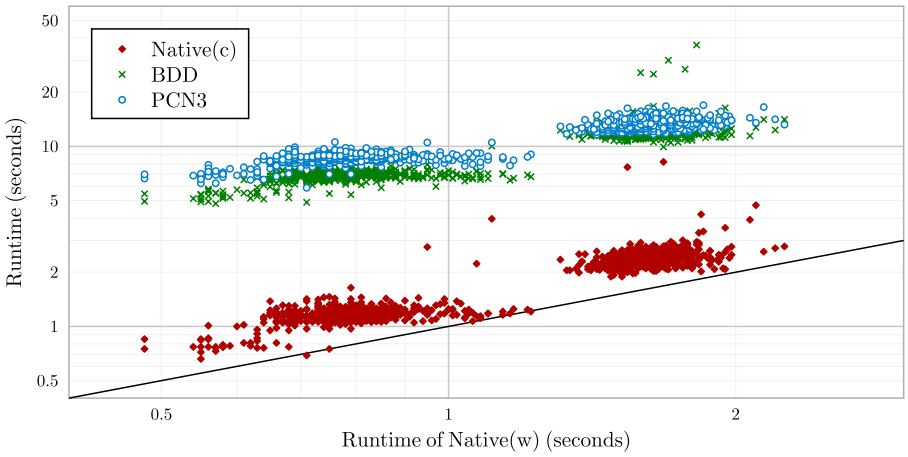

Fig. 14. Random, $n = 10^6$: Runtime for Native(w) (x-axis) vs other constraint types.

it is worth investigating why native implementations almost always produce a larger search tree in these problems. As has been done with SAT, it would also be valuable to explore randomized cardinality instances further. While cardinality constraints subsume CNF clauses, Pseudo-Boolean constraints subsume cardinality and thus it would be useful to perform an evaluation including PB implementations.

8 Acknowledgements

I would like to thank Dr. Mark Liffiton for his guidance and assistance as my research advisor. I would also like to the Illinois Wesleyan University for providing both Dr. Liffiton and Dr. Andrew Shallue the resources to fund Hyperion, the system used for my research, and of course to the two of them for allowing me use it. Finally I want to thank Dr. Albert Oliveras i Llunell and the authors of [2] for providing me with the benchmarks used in their evaluation of cardinality constraints.

References

1. SAT 2011 Competition website. http://www.cril.univ-artois.fr/SAT11/.
2. Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16:195–221, 2011.
3. K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.
4. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference*

on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99), volume 1579 of *LNCS*, pages 193–207, 1999.
5. B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.
6. Michael Codish and Moshe Zazon-Ivry. Pairwise cardinality networks. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *LNCS*, pages 154–172, 2010.
7. J.M. Crawford and A.B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.
8. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-2003)*, volume 2919 of *LNCS*, pages 502–518, 2003.
9. Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.
10. Anthony G. Frutos, Qinghua Liu, Andrew J. Thiel, Anne Marie W. Sanner, Anne E. Condon, Lloyd M. Smith, and Robert M. Corn. Demonstration of a word design strategy for DNA computing on surfaces. *Nucleic Acids Research*, 25(23):4748–4757, 1997.
11. R.J. Gardner, P. Gritzmann, and D. Prangenberg. On the computational complexity of reconstructing lattice sets from their x-rays. *Discrete Mathematics*, 202(1-3):45 – 71, 1999.
12. Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, January 2008.
13. João Marques-Silva and Inês Lynce. Towards robust CNF encodings of cardinality constraints. In *Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *LNCS*, pages 483–497, 2007.
14. João Marques-Silva and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'08)*, March 2008.
15. J.P. Marques-Silva and K.A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
16. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, 2001.
17. Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2:205–211, 1992.
18. Allen Van Gelder. mkcnf random cnf formula generator for dimacs format. "`http://users.soe.ucsc.edu/~avg/software.html`".