Grounding Count Constraints

Amir Aavani, Xiongnan (Newman) Wu, David Mitchell, Eugenia Ternovska

Abstract

There are generic applications which can be used to solve a combinatorial search problem. A common approach in these tools is to use one or more efficient general low-level solvers (such as SAT solver or ILP solver) in background. A user specifies his/her problem in a high level language and the specification is converted to the solver's input language. The process of translating a high-level language to a solver's input language is called grounding.

In our group, we had a grounding engine whose input language was a fragment of first-order logic (FO) extended with arithmetic. Although, the language was powerful enough to express all NP search problems but specifying certain problems in the language was complicated. To resolve this issue, we have added aggregates to the the syntax of engine's input language and extended the original grounding algorithm to be able to handle these constructs. To do so, we developed a method which allows us to choose a particular representation for aggregates. In this paper, we describe the six representations we have used to convert count aggregate to SAT. We compared the performance of these encodings both theoretically and experimentally. The results show that each of them performs the best in some cases and this motivates us to propose that automatic methods for selecting the encoding of each constraint at grounding time may be better than any fixed choice, and also better than having the choice made by the author of the specification. We make a preliminary suggestion toward developing such a method.

1 Introduction

An important direction of work in constraint-based methods is the development of declarative languages for specifying or modelling combinatorial search problems. These languages provide users with a notation in which to give a high-level specification of a problem (see e.g. ESSENCE [5]). By reducing the need for specialized constraint programming knowledge, these languages make the technology accessible to a wider variety of users. In our group, a logic-based framework for a specification/modelling language was proposed [9]. We undertake a research program of both theoretical development and demonstrating practical feasibility through system development.

Our tools are based on *grounding*, which is the task of taking a problem specification, together with an instance, and producing a propositional formula representing the solutions to the instance[1]. Our engine's input language is FO extended with arithmetic and aggregates. Here, we consider grounding to propositional logic, with the aim of using SAT solvers as the problem solving engine. Note that SAT is just one possibility. A similar process can be used for grounding from a high-level language to e.g. cplex, various SMT and ground constraint solvers, e.g. MINION [6], etc., which is a subject of future work. An important advantage in solving through grounding is that the speed of ground solvers improves all the time, and we can always use the best and the latest solver available.

The ultimate goal of having a high-level language is to enable naive users to encode their problems. For this purpose, it is necessary to enrich the language with the useful structures, such as COUNT and SUM aggregates. Although adding these aggregates does not increase the expressive power of the language, it makes the process of encoding of problems much easier. Currently, our grounder gets a problem description as its input and outputs a CNF, an input to SAT solver, and translates a satisfying assignment, if one is found, into a solution for the original problem. To extend the input language of the grounder, we need to be able to convert the new structures to CNF.

[1]By instance we always understand an instance of a search problem, e.g. a graph is an instance of 3-colourability.

This rest of this paper is organized as follows: In section 2, a detailed description of six translations is presented. The number of clauses and Tseiten variables generated by each of those encodings are studied in section 3. In next section, we present results of experiments that show that the performance of SAT solvers is affected by the encoding and in section 5 we describe an algorithm which can be used to select the best translation.

2 Encodings for Count

A *count-gadget*, C, is a procedure which gets a set of ground formulas, $S_F = \{f_1, \cdots, f_n\}$, and a non-negative integer, k, as its input and outputs a set of clauses, $C^k_{S_F}$, such that in all solutions for $C^k_{S_F}$ exactly k formulas out of the set S_F are true. Essentially, a count-aggregate in a first order specification can be converted to a cardinality constraints in CNF representation by introducing a new Tseiten variable for each ground formula in the aggregate. The detailed description of how a Tseiten variable can be created for a ground formula can be found [10]. In SAT context, cardinality constraints have been studied for a long time and there are several encodings for cardinality constraints which support unit propagation[8][1]. First, a high-level description of two well-known such encodings, BDD and Sorting-Network based encodings, are presented. In the rest of this section, a modified version of one of the existing encodings is discussed and then three translations, which are to the best of our knowledge are new encoding and have not previously appeared in the literature, are described.

We may use C^k_S instead of $C^k_{S_F}$ where $S = \{x_1, \cdots, x_n\}$ is a set of Tseiten variables and $S_F = \{f_1, \cdots, f_n\}$ is a set of ground formulas such that x_i is the Tseiten variable created by grounding algorithm for f_i.

2.1 Count-Gadget Through BDDs (DP)

A set of clauses encoding a cardinality constraint can be constructed based on a BDD representation of the constraint. We adopt the algorithm described in [4]. Each BDD node is an if-then-else gate. So, the constraint $|x_1, \cdots, x_n| = k$ can be described using $(n-k+1) \times k$ such nodes.

The idea we used is similar to that of BDD circuits. We inductively define F^c_r's variables using the following relations:

$$F^c_r = \begin{cases} \top & \text{if } r \text{ and } c \text{ are both zero} \\ \bot & r = 0 \text{ and } c > 0 \end{cases}$$

and

$$F^{c+1}_{r+1} = (F^c_r \wedge x_{r+1}) \vee (F^{c+1}_r \wedge \neg x_{r+1})$$

And the other inductive case is $F^0_{r+1} = F^0_r \wedge \neg x_{r+1}$

Intuitively, in each assignment, M, which $M \models F^c_r$ we have $|\{x_i | M \models x_i, i \leq r\}| = c$. i.e., in all the assignments satisfying F^c_r, out of the first r input variables, exactly c of them are true. The way F variables are described is very similar to how one fills the arrays in "dynamic programming" approaches.

Prop. 1. *If an assignment satisfies F^k_n, it satisfies exactly k formulas out of $\{x_1, \cdots, x_n\}$. So, F^k_n can be used as the output of $C(\{x_1, \cdots, x_n\}, k)$.*

Sometimes, we refer to the result of a gadget as answer produced by that gadget.

Ex. 1. *An answer to $C(\{x_1, \cdots, x_3\}, 1)$ can be produced using this method as:*

$$F^1_3 = (x_3 \wedge (\neg x_2 \wedge \neg x_1)) \vee (\neg x_3 \wedge ((x_2 \wedge \neg x_1) \vee (\neg x_2 \wedge x_1)))$$

2.2 Count-Gadget Using Sorting-Networks (SN)

A sorting network is a circuit with n input wires and n output wires consisting of a set of comparators with two input wires and two output wires. A comparator compares its two inputs and outputs the greater one into its first output wire and puts the smaller one on its second output wire. Each output of a comparator is used as an input to another comparator except those used as output wires of the sorting network itself[2]. A comparator element can be defined as a circuit with two input wires, f_1 and f_2, and two output wires, o_1 and o_2, where $o1 = f_1 \vee f_2$ and $o_2 = f_1 \vee f_2$.

If the input to a sorting network contains Z zeros and O ones, the first O outputs of the sorting network are guaranteed to contain one and the rest of wires contain zeros. Therefore, it can be concluded that there are exactly i ones in the input iff the value of i-th output wire is one while the $i+1$-th output wire is zero.

Prop. 2. *Let $x_1, \cdots, x_n$ be variables assigned to the input wires of a sorting network and $o_1, \cdots, o_n$ be the output wires. The output of the Count-Gadget which uses the sorting network is:*

$$C_S^k = \begin{cases} \neg o_1 & k\text{=}0 \\ o_k \wedge \neg o_{k+1} & k < n \\ o_n & k = n \end{cases}$$

2.3 Modification in Count-Gadget Through BDD (DP2)

In the encoding presented in 2.1, to describe F_{r+1}^{c+1} we need to create two new Tseiten variables and nine clauses. With a little modification in the meaning of the F's variable, we can reduce the number of auxiliary Tseiten variables and clauses.

Let use F_r^c to mean that in every assignment, M, that satisfies F_r^c we have $c \leq |\{x_i : 1 \leq x \leq r : M \models x_i\}|$, i.e., among the first r variables, at least c of them are true in every model that satisfies F_r^c. The following inductive relations describe F_r^c:

$$F_r^c = \begin{cases} \top & c = 0 \\ \bot & r = 0 \text{ and } c > 0 \end{cases}$$

and

$$F_{r+1}^{c+1} = (F_r^c \wedge x_{r+1}) \vee F_r^{c+1}$$

Prop. 3. *Let $S = \{x_1, \cdots, x_n\}$ and k be the inputs to the gadget and F_n^k be as described above. The following can be an answer to C_S^k:*

$$C_S^k = \begin{cases} F_n^k \wedge \neg F_n^{k+1} & k < n \\ F_n^k & k = n \end{cases}$$

2.4 Count-Gadget using Divide-and-Conquer method (DC)

The idea is to divide the variables in set $S = \{x_1, \cdots, x_n\}$ into two set $S_1 = \{x_1, \cdots, x_{\frac{n}{2}}\}$ and $S_2 = \{x_{\frac{n}{2}+1}, \cdots, x_n\}$, and then find the conditions describing how many variables from each subset is true. In next step, appropriate conditions would be merged, creating the condition about the set S. More formally, let $D_{(s,e)}^c$ be the Tseiten variable which is true in a assignment M iff $|\{x_i : s \leq i \leq e \wedge M \models x_i\}| = c$. So, $D_{(1,n)}^c$ is the necessary and sufficient condition for having exactly c variables evaluated as true out of set S . $D^c_{(s,e)}$ can be described inductively as follows:

$$D^c_{(s,e)} = \begin{cases} \bot & c > e-s+1 \\ x_s & c = 1 \text{ and } s = e \\ \neg x_s & c = 0 \text{ and } s = e \end{cases}$$

and

$$D^c_{(s,e)} = \bigvee_{i=0}^{c} D^i_{(s,(s+e)/2)} \wedge D^{c-i}_{((s+e)/2+1,e)}$$

Prop. 4. *Let $S = \{x_1, \cdots, x_n\}$ and k be the inputs to the gadget described above. $D^k_{(1,n)}$ is an answer to C^k_S. It can be proved that this encoding supports full unit-propagation.*

2.5 Binary Encoding for Count-Gadget

In this part, we present two translations to CNF for cardinality formula by using binary encodings. As we will see in next section, the number of auxiliary variables are reduced dramatically but unfortunately, neither of those two encodings support full unit-propagation.

Let $K = \lceil \log k \rceil$ be the number of bits we need to represent k in binary format. The idea is to use K binary variables to represent the number of x_i's whose value is true in the model.

We can use either incrementers or adders to describe cardinality constraints in CNF.

2.5.1 Incrementer Based Count-Gadget(BDP)

Let's assume we have an encoding for a circular incrementer which has the following property. The incrementer gets K variables, $I_K = \{i_1, \cdots, i_K\}$ as its input and outputs K variables, $O_K = \{o_1, \cdots, o_k\}$ such that:

1. If the integer value represented by $\langle i_1, \cdots, i_K \rangle$ is equal to $2^K - 1$, all o_i's are false.
2. Otherwise, the integer represented by $\langle o_1, \cdots, o_k \rangle$ is the successor of the integer represented by $\langle i_1, \cdots, i_k \rangle$.

To encode the cardinality constraint with parameters $S = \{x_1 \cdots, x_n\}$ and k, we use n arrays, N^r, each having $K+1$ variables. Given a consistent assignment M, N^r's have the following properties:

1. $M \models N^r[K+1]$ iff $2^k \leq |\{x_i : 1 \leq i \leq r \& M \models x_i\}|$.
2. $M \not\models N^r[K+1]$ and m is the integer represented by $\langle N^r[1] \cdots N^r[K] \rangle$ under assignment M iff $m = |\{x_i : 1 \leq i \leq r \& M \models x_i\}|$.

N^r arrays, $r \geq 1$, satisfy the following relations:

$$N^{r-1}[K+1] \vee (x_r \wedge \bigwedge_{i=1}^{K} N^{r-1}[i]) \Rightarrow N^r[K+1]$$

$$\neg x_r \Rightarrow \bigwedge_{i=1}^{K} (N^r[i] \Leftrightarrow N^{r-1}[i])$$

$$x_r \Rightarrow N^r = INC(N^{r-1}[1..K])$$

and $N^0 = \langle \bot, \cdots, \bot \rangle$.

Prop. 5. *Let $S = \{x_1, \cdots, x_n\}$ and k be the inputs to the gadget described above and $K = \lceil \log k \rceil$. If k has a binary representation like $\lceil k_1 \cdots k_K \rceil$, the following is an answer to C^k_S:*

$$\neg N^n[K+1] \wedge \bigwedge_{i=1}^{K} (N^n[i] \Leftrightarrow k_i = 1)$$

2.5.2 Count-Gadget Using Binary Adder(BDP)

Adders can be used to encode the cardinality constraints. Let's assume we have an encoding for an adder which has the following property. The incrementer gets two arrays, each containing K variables, $I = \{i_1, \cdots, i_K\}, I' = \{i'_1, \cdots, i'_K\}$ as its input and outputs a pair $\langle O_K, Overflow\rangle$ where O_k is an array of K variables, $O_K = \{o_1, \cdots, o_k\}$ and Overflow is a single variable (flag) such that (if n_I and $n_{I'}$ are the integer corresponding two the two Boolean arrays I and I', respectively):

1. if $n_I + n_{I'} < 2^K$, the Overflow would be false and the O array would have the binary representation of $n_I + n_{I'}$.
2. if $n_I + n_{I'} \geq 2^K$, the Overflow would be true and the O array would have the binary representation of $(n_I + n_{I'}) \mod 2^K$.

To encode the cardinality constraint with parameters $S = \{x_1 \cdots, x_n\}$ and k, we use about $2n$ arrays[2], N^e_s, each having $K+1$ variables. Given a consistent assignment M, N^e_s's have the following properties:

1. $M \models N^e_s[K+1]$ iff $2^k \leq |\{x_i : 1 \leq i \leq r \& M \models x_i\}|$.
2. $M \not\models N^e_s[K+1]$ and m is the integer represented by $\langle N^e_s[1] \cdots N^e_s[K]\rangle$ under assignment M iff $m = |\{x_i : 1 \leq i \leq r \& M \models x_i\}|$.

N^e_s arrays, $e > s$, satisfy the following relations:

$$\langle N^e_s[1..K], N^e_s[K+1]\rangle = ADD(N_s^{(s+e)/2}[1..K], N^e_{(s+e)/2+1}[1..K])$$
$$N_s^{(s+e)/2}[K+1] \vee N^e_{(s+e)/2+1}[K+1] \Rightarrow N^e_s[K+1]$$

and $N^s_s[1] \Leftrightarrow x_s$, $\bigwedge_{i=2}^{K} \neg N^s_s[i]$.

Prop. 6. *Let $S = \{x_1, \cdots, x_n\}$ and k be the inputs to the gadget described above and $K = \lceil \log k \rceil$. If k has binary representation like $\lceil k_1 \cdots k_K \rceil$, the following is an answer to C^k_S:*

$$\neg N^n_1[K+1] \wedge \bigwedge_{i=1}^{K} (N^n_1[i] \Leftrightarrow k_i = 1)$$

[2] We will determine the exact number in next section

3 Size of Generated CNF

In this section, In this section, we describe the CNF formulas generated by each gadget in terms of several measures, namely, number of clauses, number of variables, number of literals, average clause size and depth of CNF. By depth of a CNF, we mean the depth of corresponding Boolean circuit.The results are summerized in Table 1.

In DP method, to describe F_r^c one needs to introduce and describe all F_j^i's where $0 \leq i \leq c$ and $0 \leq j \leq r$. In most cases, describing each F_j^i needs disjunction of two conjunctions. The normal way of expressing such a thing in CNF is to introduce a new variable for each conjunction and expressing F_j^i based on the two new variables.

Prop. 7. *To convert a cardinality constraint with n input variables whose result should be m, $C^m_{\{x_1,\cdots,x_n\}}$, to CNF, DP method uses $2nm$ auxiliary variables and generates $9nm$ clauses. There are $21nm$ literals used in corresponding CNF. Depth of the circuit resulting from the generated CNF would be* $2\min(c,r)$.

The second approach, SN, is based on creating a sorting network for n variables. There is not any efficient construction for sorting network which uses $\theta(n\log n)$ comparators, but there are some which need $\theta(n\log^2 n)$ comparators and have a good performance in practice, for example "Bitonic Sorters"[11] and "Insertion and Selection Networks". Each comparator can be described in CNF by introducing two auxiliary variables, for its output, and using 6 clauses describing the relation between inputs and outputs.

Prop. 8. *To convert a cardinality constraint with n input variables whose result should be m, $C^m_{\{x_1,\cdots,x_n\}}$, to CNF, SN method uses $2n\log^2 n$ auxiliary variables and generates $6n\log^2 n$ clauses. There are $14n\log^2 n$ literals used in corresponding CNF. Depth of the circuit resulting from the generated CNF would be* $\log^2 n$.

In first look, one may think that DC method to describe $S^c_{\{x_1,\cdots,x_n\}}$ needs to create n^2c D' variables. But there is an implementation which creates just $(n-1)c$ of D variables[3].

Prop. 9. *The relations between the D variables can be presented using nc^2 auxiliary variables and $3nc^2$ clauses. In the generated CNF file has $5nc^2$ literals. Also, this encoding needs more variables and literals, but the depth of the corresponding circuit is* $\log n$ *which is smaller than both previous methods.*

The modification in DP reduces the number of auxiliary variables and clauses by a constant factor.

Prop. 10. *To convert a cardinality constraint $C^m_{\{x_1,\cdots,x_n\}}$, to CNF, DP2 method uses $nm+1$ auxiliary variables and generates $6nm+1$ clauses. There are $14nm+2$ literals used in corresponding CNF. Depth of the circuit resulting from the generated CNF would be* $2\min(c,r)$.

Adders and incrementers can be encoded by circuits, and so in CNF, very easily. "Ripple carry adder" and "Carry skip adder"[3] are two well-known circuits which adds two binary digits. Although every adder can be used as an incrementer, an incrementer can be implemented more efficiently from scratch, too. In this analysis, we have used the most naive encodings, i.e., circuits, for adders and incrementers.

There is an implementation for a parallel K-bit incrementer with a constant depth which needs $\theta(K^2)$ clauses, $\theta(K^2)$ literals and K auxiliary variables. To describe a count-aggregate using BDP, n incrementers and some other clauses need to be generated.

[3]That implementation is a normal recursive procedure

Table 1: Comparison among the # of Clauses, # of Auxiliary Variables, # of Literals and Depth of CNF generated by each gadget, for constraint in the form of $C^{m}_{\{x_1,\cdots,x_n\}}$

Encoding	# of Clauses	# of Auxiliary Vars	# of Literals	Depth
DP	$9nm$	$2nm$	$21nm$	$2\min(n,m)$
DP2	$6nm$	nm	$14nm$	$2\min(n,m)$
DC	$3nc^2$	nc^2	$5nc^2$	$\log n$
SN	$6n\log^2 n$	$2n\log^2 n$	$14n\log^2 n$	$\log^2 n$
BDP	$\theta(n\log^2 m)$	$\theta(n\log m)$	$\theta(n\log^2 m)$	n
BDC	$\theta(n\log m)$	$\theta(n\log m)$	$\theta(n\log^2 m)$	$\log m\log n$

Prop. 11. *The number of clauses, variables and literals can be estimated by $\theta(nK^2)$, nK and $n\theta(K^2)$, respectively. Depth of corresponding circuit would be n.*

There is an encoding of an K-bit by K-bit adder in CNF which generates $\theta(K)$ clauses, $\theta(K^2)$ literals using $\theta(K)$ auxiliary variables. Depth of corresponding circuit is K.

Prop. 12. *The BDC approach uses, roughly speaking, n adders. So, the BDC generates $\theta(nK)$ clauses,, $\theta(nK)$ auxiliary variables and $\theta(nK^2)$ literals. Depth of corresponding circuit is $K\log n$ as there are at most $\log n$ sequential adders in the encoding.*

In table 1, the information about these six gadgets are presented.

4 Experiments

To compare the performance of gadgets, we have used three different problems which can be encoded using count. We generated a set of random instances for each problem.

1. Blocked N-queens: A blocked N-queens problem asks for placing N queens in the non-blocked cells of an $N \times N$ board, where some of its cells are already blocked, such that none of them are able to capture any other using the standard chess queen's moves, I.e. no two queens share the same row, column, or diagonal.

2. Rectangle Coloring: In rectangle coloring problem, we are asked to color each cell of an N by M rectangle such that there are a certain number of black cells in each row and each column. If one can forces some cells to be white, the problem would be an NP-complete problem [4]. In fact, given N, M, a partial coloring of the cells and two arrays of integers, $R = \{r_1, \cdots, r_n\}$ and $C = \{c_1 \cdots, c_m\}$, the problem asks for finding a coloring of the cells such that there are r_i black cells in the i-th row and c_i black cells in the i-th column.

3. Randomly Generated Cardinality Formulas: Given a set of variables $S = \{x_1, \cdots, x_n\}$ and a set of constraints $C = \{C_1, \cdots, c_m\}$ in the following format, the problem asks to find an assignment satisfying all the m constraints.
$$C_i : |S_i| = k_i$$
where $S_i \subset S$ and $0 \leq k_i \leq |S_i|$.

[4]An instance of set cover problem can be reduced to an instance of this problem

The sat solver used to solve all the instances is MiniSat version 2.0. Instances are generated randomly as follows:

1. Blocked N-queens: For each n, $4 \leq n \leq 40$, 100 different random instances are used.

2. Rectangle Coloring: For all n and m, such that $4 \leq n \leq 30$, and $4 \leq m \leq n$, 100 different random instances are created. The R and C arrays are selected such that $\sum_i r_i = \sum_i c_i$. And then, we fix the color of some cells of the board to either white or black.

3. Randomly Generated Cardinality Formulas: For each n and m, from 4 to 50, 100 different random instances are generated. For creating an instance, given n and m, we produce m pairs of $\langle S_i, k_i \rangle$ satisfying the condition described in problem statement.

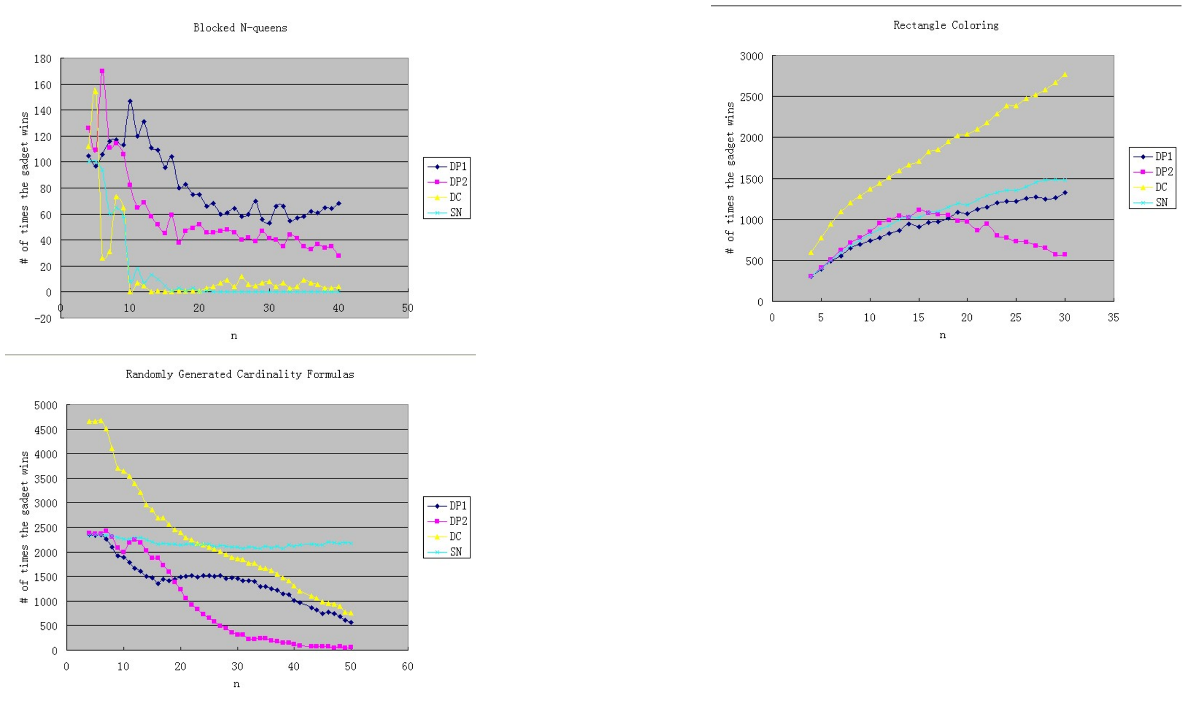

Figure 1: Number of times each gadget performs the best (or it is one of the best if there is a tie).

As it is discussed previously, there is not a well-defined measure for determining the best encoding. On an instance, we say a gadget performs the best if the amount of cpu-time, in milliseconds, sat-solver needs to solve the CNF created using that gadget is minimum. For each instance, we ranked all the four gadgets and count how many times each was the winner.

The values in Figure 1show that there is no gadget which outperforms the other one.

5 Automatic Gadget Selection

As the experimental results in previous section show, each gadget performs better in a certain case which depends on n and k. When a user develops a specification for a certain problem, he/she may not know about the properties of instances. For example, consider the following encoding for ”blocked n queen”:

$$\forall r : \#_c(B(r,c) \wedge Q(r,c)) = 1 \quad (1)$$

$$\forall c : \#_r(B(r,c) \wedge Q(r,c)) = 1 \quad (2)$$

$$\forall n : \#_r(B(r,r+n) \wedge Q(r,r+n)) = 1 \quad (3)$$

$$\forall n : \#_r(B(r,r-n) \wedge Q(r,r-n)) = 1 \quad (4)$$

$$\forall n : \#_c(B(c,c+n) \wedge Q(c,c+n)) = 1 \quad (5)$$

$$\forall n : \#_c(B(c,c-n) \wedge Q(c,c-n)) = 1 \quad (6)$$

It can be the case that for some of rows there are very few un-blocked cells while for some others, there are many un-blocked cells. Consider the **square coloring** problem described in previous section. Something similar may happen for that problem, too. The results in table 1 suggest that using the same gadget to encode all the cardinality formula is not a good idea. There are other problems for which a user can not specify the best gadget with out observing the contents of the instance data.

Having an algorithm which can automatically select the best gadget can be very helpful. The main challenge in the way is that how one defines the best gadget. Is it the one which produces the least number of clauses? Or the one with smaller search space is better? One can argue that the running time of SAT solver on the generated CNF is the most important factor while the others may suggest the maximum amount of memory used by SAT solver as the most important parameter? Some efforts are done in order to estimate the running time of SAT solvers like, [7]. In the setting described here, we can not use such cost functions as we need an estimation for running time before starting the SAT solving phase.

It is hard to select the best gadget for a cardinality formula as soon as we meet one. There may be several cardinality formulas with the same set of variables. Making local choices separately may not achieve the best performance in this case. Assume we have a cardinality formula $|x_1,\cdots,x_n| = k$, and we make a local choice to use ”Sorting Network” gadget. We may have another cardinality formula with the same set of variables $|x_1,\cdots,x_n| = k'$, and we decide to use DP Gadget. We create a lot of redundant clauses by running different gadgets on formulas with the same set of variables. It would be good if we could group formulas with the same set of variables and select a suitable gadget for them. Then, we need to run the appropriate gadget only once to generate CNF for cardinality formulas with the same set of variables[5].

To achieve this, we can postpone generating CNF for the cardinality formulas and use a new Tseiten variable to represent the result of each such cardinality formula. At the end, we have the list of all the cardinality formulas and their corresponding Tseiten variables, $\langle v_i, |\{x^i_1 \cdots, x^i_{n_i}\}| = m_i\rangle$ where for all structure M:

$$M \models v_i \Leftrightarrow M \models |\{x^i_1,\cdots,x^i_{n_i}\}| = m_i$$

By sorting the list, the cardinality formulas can be grouped such that those who have the same set of variables are placed in the same group. And using the table 1 one can decide which gadget may perform the best.

6 Conclusion & Future Works

In this paper, we presented six encodings for count aggregate (cardinality constraints) and analyzed their CNF output files. Based on the theoretical analyze presented in section 3, it can be expected that

[5]Although we did not describe how this is possible but it is not hard to modify the described gadgets to do so

there are certain values for n and k such that each gadget outperforms the other gadgets. Considering the fact that the first four gadgets supports full unit-propagation, it sounds logical to expect a trend like the following for a fixed value of n:

1. For small value of k, $k \ll n$: DP (DP2) performs the best.
2. By increasing the value of k, after a certain point, DC outperforms DP.
3. When k is large enough, around n, SN becomes the best encoding.

The result of experiments confirms this claim.

We also addressed the need of having an algorithm which automatically selects the best gadget based on a certain cost function. The experiments show that different gadgets have different performances which are a support for this idea.

We have not implemented BDP and BDC gadgets yet, and implementing them and comparing their performance with the SN is the next step in our work list.

References

[1] BAILLEUX, O., AND BOUFKHAD, Y. Efficient CNF encoding of Boolean cardinality constraints. *Lecture notes in computer science* (2003), 108–122.

[2] BATCHER, K. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference* (1968), ACM, pp. 307–314.

[3] BRUCE, J., THORNTON, M., SHIVAKUMARAIAH, L., KOKATE, P., AND LI, X. Efficient adder circuits based on a conservative reversible logic gate. In *Proc. IEEE Computer Society Annual Symposium on VLSI* (2002), Citeseer, pp. 83–88.

[4] EÉN, N. *SAT Based Model Checking.* PhD thesis, 2005.

[5] FRISCH, A. M., GRUM, M., JEFFERSON, C., HERNANDEZ, B. M., AND MIGUEL, I. The design of ESSENCE: a constraint language for specifying combinatorial problems. In *Proc. IJCAI'07* (2007).

[6] GENT, I., JEFFERSON, C., AND MIGUEL, I. Minion: A fast, scalable, constraint solver. In *ECAI 2006: 17th European Conference on Artificial Intelligence, August 29-September 1, 2006, Riva del Garda, Italy: including Prestigious Applications of Intelligent Systems (PAIS 2006): proceedings* (2006), Ios Pr Inc, p. 98.

[7] HAIM, S., AND WALSH, T. Online estimation of sat solving runtime. *Lecture Notes in Computer Science 4996* (2008), 133–138.

[8] MARQUES-SILVA, J., AND LYNCE, I. Towards robust CNF encodings of cardinality constraints. *Lecture Notes in Computer Science 4741* (2007), 483.

[9] MITCHELL, D., AND TERNOVSKA, E. A framework for representing and solving NP search problems. In *Proc. AAAI'05* (2005).

[10] MOHEBALI, R. A method for solving np search based on model expansion and grounding. Master's thesis, Simon Fraser University, 2006.

[11] PATERSON, M. Improved sorting networks with O (log N) depth. *Algorithmica 5*, 1 (1990), 75–92.