Efficient CNF Encoding for Selecting 1 from N Objects

Will Klieber[1], Gihwon Kwon[2]

Department of Computer Science, Carnegie Mellon University[1]
wklieber@andrew.cmu.edu

Department of Computer Science, Kyonggi University[2]
khkwon@kyonggi.ac.kr

Abstract. In many boolean-satisfiability problems, one must encode the constraint that at most one of n propositional variables is true. With a naïve encoding, this requires $O(n^2)$ CNF clauses. We present a flexible alternative encoding that only requires $O(n)$ clauses, at the expense of $O(n)$ extra variables. The proposed encoding technique also allows efficient encoding of canonical-ordering constraints that can aid in the determination of unsatisfiable problem instances. Sample applications are given for the pigeon-hole problem and Sudoku puzzles.

1 Introduction

In many boolean-satisfiability (SAT) problems, one must encode the constraint that exactly one of n propositional variables is true. This constraint is usually broken down into two constraints: (1) At least one variable is true, and (2) At most one variable is true. The "at least one" constraint is simple to encode as a single clause in CNF form, but the "at most one" constraint is more difficult. The traditional way of handling the "at most one" constraint, if performance is not an issue, to explicitly require that every pair of variables have at least one false variable. However, this requires enumerating all possible pairs of variables, leading to $O(n^2)$ clauses.

This paper presents a flexible and efficient encoding for "at most one" constraints. If there are a large number of variables, our approach requires $O(n)$ clauses and $O(n)$ extra variables. Even for a small number of variables, our approach never performs worse than the naive encoding; it will automatically and naturally reduces to the naïve encoding if there are too few variables (less than 6) to overcome the overhead of our approach.

This technique is applied to a variant of the pigeon-hole problem and to the solving of Sudoku puzzles. For the pigeon-hole problem, our approach additionally allows for the efficient encoding of canonical ordering constraints, which greatly reduce the time needed to determine unsatisfiable instances. For the Sudoku application, our approach, combined with a preprocessing step that quickly eliminates a large class of variables and clauses, is able to handle board sizes up to at least 144×144.

The rest of the paper is organized as follows. First, we describe our approach, which we call "commander-variable encoding", in Section 2. Next, we describe the applications to the pigeon-hole problem and to Sudoku puzzles in Sections 3 and 4.

We discuss related work in Section 5, and we conclude in Section 6.

2 Commander-Variable Encoding

Suppose that we have a set of propositional variables $X = \{x_1, \ldots, x_n\}$, and we desire exactly one of them to be true. In the naïve encoding, each variable must 'talk' with every other variable. That is, each variable must appear in a clause with every other variable. We can formalize this with the following functions that return a set of clauses:

$$\mathit{NaïveAtLeastOne}(X) = \bigvee_{i=1}^{n} x_i$$

$$\mathit{NaïveAtMostOne}(X) = \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} (\neg x_i \vee \neg x_j)$$

$$\mathit{NaïveExactlyOne}(X) = \mathit{NaïveAtLeastOne}(X) \wedge \mathit{NaïveAtMostOne}(X)$$

To reduce the number of clauses, we can divide the variables into groups. We assign a new variable, called "a commander variable", to each group. This commander variable is to be true if (at least) one of the variables in its group is true; otherwise it is to be false. In the commander-variable method, the original variables do not need to 'talk' directly to any other variables that are not in the same group; instead, the commander variables act as proxies between original variables in different groups.

To describe the commander-variable encoding more precisely, let us introduce additional notation. Let the set of propositional variables $X = \{x_1,\ldots,x_n\}$ be divided into m disjoint subsets G_1 through G_m. The commander node of group G_i is labeled "c_i". Using this notation, the logic for the commander method can be encoded in CNF form as follows; a running example is given for the case where the variables $\{x_1, x_2, x_3\}$ are grouped together and their commander variable is c_1:

1. **At most one variable in a group can be true.** This is encoded by the traditional pair-wise method that was mentioned earlier. For each group G_i, we encode the following clauses:

$$\bigwedge_{x_j \in G_i} \bigwedge_{x_k \in G_i, k<j} (\neg x_j \vee \neg x_k)$$

Example: $(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3)$.

2. **If the commander variable of a group is true, then at least one of the variables in the group must be true.** (To encode the "at most one" rather than the "exactly one" constraint, omit this step.) For each group G_i, we encode the following clause:

$$\neg c_i \vee \bigvee_{x_j \in G_i}^{n} x_j$$

Example: $c_1 \Rightarrow (x_1 \vee x_2 \vee x_3)$, which reduces to $\neg c_1 \vee x_1 \vee x_2 \vee x_3$ in CNF.

3. If the commander variable of a group is false, then none of the variables in the group can be true. For each group G_i, we encode the following clauses:

$$\bigwedge_{x_j \in G_i} (c_i \vee \neg x_j)$$

Example: $\neg c_1 \Rightarrow (\neg x_1 \wedge \neg x_2 \wedge \neg x_3)$, which reduces to $(c_1 \vee \neg x_1) \wedge (c_1 \vee \neg x_2) \wedge (c_1 \vee \neg x_3)$ in CNF.

4. Exactly one of the commander variables is true. This can be encoded either by the pair-wise method or by a recursive application of the commander method. For the pair-wise method, we encode the following clauses, where m is the number of groups:

$$(c_1 \vee c_2 \vee \ldots \vee c_m) \wedge \bigwedge_{i<m} \bigwedge_{j<i} (\neg c_i \vee \neg c_j)$$

In the case of a recursive application, a hierarchy of commander variables is formed, as depicted in Fig. 1.

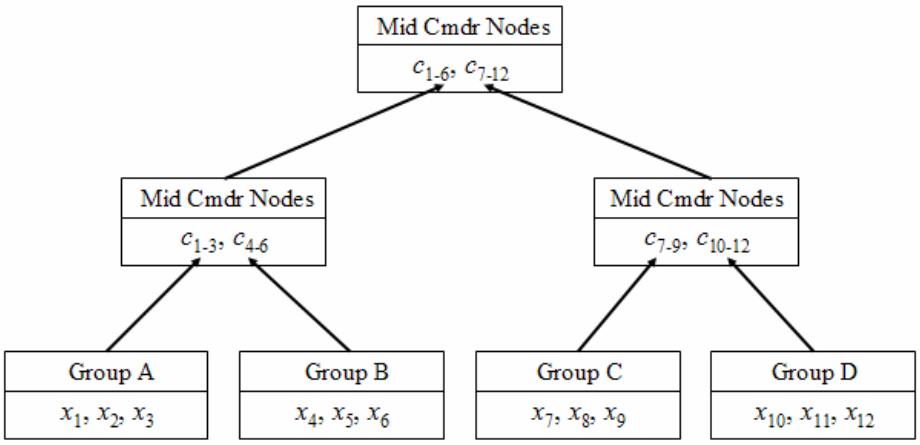

Figure 1. Hierarchy of commander variables

Let's consider how many clauses are required for each group. Note that Constraint 1 above requires $n*(n\text{-}1)/2$, where n is the number of variables in the group. Constraint 2 requires 1 clause, and Constraint 3 requires n clauses. (A commander variable is not considered to be 'in' the group that it commands.) Thus, the total number of clauses per group is $(n*(n+1)/2)+1$. (This does not include the clauses required by Constraint 4 to ensure that exactly one of the commander variables is true.)

2.1 Analysis

The commander encoding is flexible in that any method can be used to group the variables. However, to analyze the performance of the commander approach (in terms of many clauses and extra variables are required), let us consider a grouping method wherein the variables are, at each stage of the hierarchy, divided into groups of k variables. For example, k=2, we get binary tree shown in Fig. 2. From the diagram, it is clear that the number of groups, in the asymptotic limit of a large number of variables n, is:

$$\begin{aligned}
&\sum_{i=1}^{\infty}(n/k) = (n/k)+(n/k^2)+(n/k^3)+\cdots \\
&= (1/k+1/k^2+n/k^3+\cdots)n \\
&= \left(\frac{1}{1-1/k}-1\right)n \\
&= \left(\frac{1/k}{1-1/k}\right)n
\end{aligned}$$

The number of extra variables is equal to the number of groups. To get the number of clauses, we multiply this by the number of clauses per group ($(k*(k+1)/2)+1$ from the previous section), getting a total of

$$\left(\frac{(k+1)/2+(1/k)}{1-1/k}\right)n$$

clauses. It turns out that the best choice for minimizing the number of clauses is k=3. In this case, the number of clauses is 3.5n and the number of extra variables is n/2.

Now, let us examine the case in which there are a small number of variables. In order to benefit from the commander method, we need at least 6 variables. In that case, we divide the variables into 2 groups of 3 variables. Each group requires (1/2)(3+1)(3)+1 = 7 clauses. Normally, to ensure that exactly one of the two commander variables is true, we would need 2 additional clauses, bringing the total to 16. However, since there are only two top-level commander variables, we can encode the second as the negation of the first, so that it is automatically true that exactly one of commander variables is true. Thus, we only need 14 clauses. This compares to 16 clauses for the naïve encoding *NaïveExactlyOne*($\{x_1,\ldots,x_6\}$).

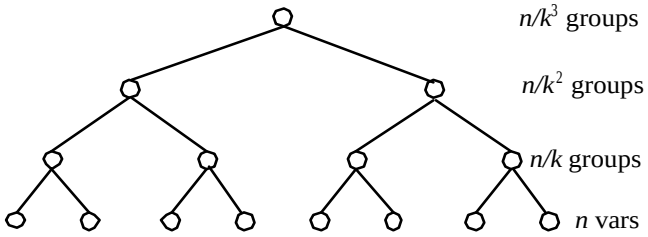

Figure 2. Binary tree of groups

2.2 Algorithm Details

Fig. 3 shows an algorithm for the commander-variable encoding. The function is passed a nested list of propositional variables (or literals). The nesting determines how the variables are grouped the under the commander nodes. Since there is the wrapper function, an application may just call the function *CmdrExactlyOne* with the set of propositional variables and receives the set of clauses in CNF from the function. The grouping function is defined in Fig. 4 and called from *CmdrExactlyOne*.

```
Algorithm CmdrExactlyOne(Subords, CmdrVar) =
input
   Subords;          -- a nested list of subordinates.
                     -- a subordinate is a variable or a list of subordinates.
   CmdrVar;          -- the variable number for the commander variable.
begin
    φ = [ ];         -- clauses to be returned.
   ClauseVars = [ ]; -- variables in the clauses (2D array).
   for i = 0 to len(Subords)
      if (IsPropVar(Subord[i])) then   -- if the subord is a propositional variable.
         ClauseVars[i] = Subord[i];
      else
         assert(IsList(Subord[i]));   -- here the subord must be a list of subords.
         ClauseVars[i] = VarAlloc();
         φ = φ ∧ CmdrExactlyOne(Subord[i], ClauseVars[i]);
      endif
   endfor
   if (CmdrVar != 0) then
      ClauseVars += Negate(CmdrVar);
   endif
   φ = φ ∧ NaïveExactlyOne(ClauseVars);
   return φ;
end

-- Wrapper Function
Algorithm CmdrExactlyOne(Subords) =
input
   Subords;     -- a set (or list) of subordinates.
begin
   return CmdrExactlyOne(GroupVars(Subords), VarAlloc());
end
```

Figure 3. Commander encoding algorithm

```
Algorithm GroupVars(Vars, MaxSize) =
input
  Vars;          -- a list of variables to be grouped into a hierarchy.
begin
  NumVars = len(Vars);
  if (NumVars <= MaxSize) then
    return Vars;
  endif
  ret = [ ];      -- group to be returned.
  NumGr = NumVars / MaxSize;    -- truncating integer division.
  for i = 0 to NumGr – 1
    ret[i] = Vars[ i∗NumVars / NumGr  ..  (i+1)∗NumVars / NumGr – 1]
    -- Note: “SomeList[first .. last]” denotes a slice of the list.
  endfor
  return GroupVars(ret, MaxSize);
end
```

Figure 4. Grouping function

3 Pigeon-hole Problem with Proposed Encoding

This section applies the commander encoding to the pigeon-hole problem and aims to show its efficiency in handling the problem “select one from a certain set of objects”. Another purpose is to show its flexibility because the commander encoding is easily combined with other techniques, such as a canonical ordering in this case, and putting the two together results in better performance in solving the pigeon-hole problem with SAT solvers.

The pigeon-hole problem asks whether n pigeons can fit in m holes [1]. The constraints for this problem can be written as follows:

- Each pigeon must be in exactly one hole.
- Each hole must have at most one pigeon.

To express the above rules as CNF clauses for SAT solvers, they must be expanded out for each particular pigeon and hole. Let $x_{i,j}$ be a propositional variable that represents whether pigeon i occupies hole j. Using this notation, the constraints can be rewritten as follows:

- Given a pigeon i, there is exactly one hole j for which $x_{i,j}$ is true.
- Given a hole j, there is at most one pigeon i for which $x_{i,j}$ is true.

These constraints can be encoded as a set ϕ of CNF clauses whose conjunction is unsatisfiable if and only if at least one hole must contain more than one pigeon in order for each pigeon to be in a hole. The naïve encoding in Fig. 5 can be used for encoding the problem into CNF clauses. It takes an n-in-m problem and generates CNF clauses with the naïve functions *NaïveAtMostOne*, *NaïveExactlyOne* defined in the previous section:

Algorithm *PigeonholeToCNF*(n, m) =
input
  n; -- number of pigeons.
  m; -- number of holes.
begin
  ϕ = [ ]; -- clauses to be returned.
  $X = \{x_{i,j} \mid 1 \le i \le n, 1 \le j \le m\}$;
  for i = 1 to n
    $\phi = \phi \wedge NaïveExactlyOne(\{x_{i,j} \in X \mid 1 \le j \le m\})$;
  endfor
  for j = 1 to m
    $\phi = \phi \wedge NaïveAtMostOne(\{x_{i,j} \in X \mid 1 \le i \le n\})$;
  endfor
  return ϕ;
end

Figure 5. Naïve encoding for the n-in-m pigeonhole problem

where the notation $\{x \in X \mid cond_1, cond_2, \ldots\}$ designates the subset of propositional variables $x \in X$ for which every condition holds true. If there are more pigeons than holes, then the problem is unsatisfiable. Table 1 shows experimental results with the naïve encoding on differently-sized unsatisfiable problems. The experiments were done with MiniSAT [2] and performed on a 1.10GHz Intel Pentium M with 1.25 GB of memory. In our experiments, the naïve encoding just handled up to the 11-in-10 problem in which 140 seconds were taken. However, the SAT solver failed for the 12-in-11 problem within the pre-defined time limitation of 600 seconds.

size		naïve			commander			commander + ordering		
n	m	variables	clauses	time	variables	clauses	time	variables	clauses	time
5	4	20	75	0.02	29	76	0.01	29	88	0.01
6	5	30	141	0.02	41	126	0.01	41	151	0.01
7	6	42	238	0.04	67	206	0.02	67	248	0.01
8	7	56	372	0.11	101	299	0.04	101	355	0.02
9	8	72	549	0.69	123	390	0.16	123	470	0.01
10	9	90	775	3.40	165	502	0.42	165	610	0.02
11	10	110	1,056	140.39	215	640	1.88	215	750	0.02
12	11	132	1,398	timeout	247	770	8.20	247	913	0.02
13	12	156	1,807	timeout	293	924	40.63	293	1,104	0.02
130	129	16,770	2,155,075	timeout	37,231	119,528	timeout	37,231	142,103	1.42

Table 1. Experimental results for unsatisfiable pigeon-hole problems ($n > m$)

The commander encoding was used to solve the large problems by replacing the naïve functions *NaiveExactlyOne* and *NaiveAtMostOne* with the commander ones *CmdrExactlyOne* and *CmdrAtMostOne*, respectively. As you see from the second column in Table 1, with the commander encoding, the SAT solver resolved the 11-in-10 problem within 2 seconds. In addition, it could solve the larger problems up to the

13-in-12 problem.

For large values of n and m, however, it takes a long time for SAT solvers to ascertain that the problem is unsatisfiable, due to the large search space. In contrast, we as human beings can immediately see that the problem is unsatisfiable by using the symmetry:

- We arbitrarily place 12 of the pigeons in the 12 holes. Then there is no room for the 13^{th} pigeon.
- Since the pigeons are interchangeable from the viewpoint of the problem constraints, we can immediately see that no permutation of the 12 placed pigeons will leave room for the 13^{th} pigeon.

SAT solvers, on the other hand, do not have the ability to use this symmetry. However, if we can detect the symmetry in a problem instance, then we can reduce the state space that needs to be explored. We do this by imposing a canonical ordering on the pigeons. Any decision branch that violates the canonical ordering is abandoned as soon as the SAT solver realizes that the canonical ordering is violated.

An example of a canonical ordering used here is as follows: Pigeon i must be placed in a hole in front of the hole of pigeon i+1 (i.e., the pigeon in hole j must have a smaller number than that of the pigeon in hole j+1). With the proposed commander encoding, however, it is easy to efficiently encode inequality relations such as *less than* and *greater than*. The commander encoding and the canonical ordering are combined in Fig. 6 which makes MiniSAT work efficiently. Due to space limitations, we do not explicitly describe in detail the canonical-ordering encoding, but the number of clauses generated is of order $O(n \log n)$.

Algorithm *PigeonholeToCNF*(n, m) =
input
  n; -- number of pigeons.
  m; -- number of holes.
begin
  ϕ = [ ]; -- clauses to be returned.
  $X = \{x_{i,j} \mid 1 \le i \le n,\ 1 \le j \le m\}$;
  for i = 1 to n
    $\phi = \phi \wedge CmdrExactlyOne(\{x_{i,j} \in X \mid 1 \le j \le m\})$;
  endfor
  for j = 1 to m
    $\phi = \phi \wedge CmdrAtMostOne(\{x_{i,j} \in X \mid 1 \le i \le n\})$;
  endfor
  for i = 1 to n-1
    $\phi = \phi \wedge CanonicalOrder(i)$;
  endfor
  return ϕ;
end

Figure 6. The commander encoding with the canonical ordering

The last column in Table 1 shows the 13-in-12 problem was solved in a millisecond.

(Note that the commander encoding alone took 41 seconds.) Even the 130-in-129 problem was solved in under 2 seconds. The commander encoding with the canonical ordering also showed good performance for satisfiable cases ($n < m$) shown in Table 2.

In summary, the commander encoding with the canonical ordering shows much better performance than the standard naïve encoding used for pigeon-hole CNF files in SATLIB [3] in terms of both the number of clauses and SAT solving time.

size		naïve			commander			commander + ordering		
n	m	variables	clauses	time	variables	clauses	time	variables	clauses	time
4	5	20	74	0.02	29	74	0.02	29	89	0.03
5	6	30	140	0.02	41	124	0.01	41	152	0.02
6	7	42	237	0.03	67	198	0.01	67	238	0.01
7	8	56	371	0.01	101	298	0.01	101	358	0.02
8	9	72	548	0.01	123	389	0.01	123	473	0.01
9	10	90	774	0.02	165	510	0.02	165	598	0.01
10	11	110	1,055	0.02	215	638	0.02	215	755	0.01
11	12	132	1,397	0.02	247	768	0.02	247	918	0.02
12	129	156	1,806	0.01	293	922	0.02	293	1,098	0.01
129	130	16,770	2,155,074	4.93	37,231	119,526	6.82	37,231	142,182	1.60

Table 2. Experimental results for satisfiable pigeon-hole problems ($n < m$)

4 Sudoku Puzzle with Efficient Encoding

This section also applies the commander encoding to Sudoku puzzles and shows its efficiency in handling the problem of selecting exactly one object out of a set of n objects. The rules of Sudoku can be written as follows [4]:

- Each cell has exactly one digit.
- Each row has exactly one of each digit.
- Each column has exactly one of each digit.
- Each block has exactly one of each digit.

As explained in Section 2, the condition “exactly one” is translated as “at least one and at most one” for CNF-based SAT solvers. Due to the symmetry in Sudoku, it is not logically necessary to explicitly encode both the “at least one” and the “at most one” conditions for each of the 4 rules above. However, in order for SAT solvers to work efficiently, in practice it is necessary to explicitly encode all 8 conditions [5].

To express the above rules as CNF clauses for SAT solvers, they must be expanded out for each particular cell, row, column, block, and/or digit. Let $x_{r,c,d}$ be the propositional variable that denotes whether the cell in row r and column c contains the digit d. Using this notation, the above rules can be rewritten as follows:

- Given a row r and a column c, there is exactly one digit d for which $x_{r,c,d}$ is true.
- Given a row r and a digit d, there is exactly one column c for which $x_{r,c,d}$ is true.
- Given a column c and a digit d, there is exactly one row r for which $x_{r,c,d}$ is true.

- Given a block b and a digit d, there is exactly one index i for which $x_{r,c,d}$ is true, where r and c are the row and column numbers of i^{th} cell of block b.

Notice that each instance of the above expansions corresponds to the following form: "out of a certain set of n propositional variables, exactly one is true". It would be easy to write a naïve encoding algorithm that takes a Sudoku puzzle represented by a 2D matrix M and generates a set ϕ of CNF clauses which is satisfiable if and only if the puzzle has a solution:

Algorithm *SudokuToCNF*(M) =
input
  $M[n][n]$; -- 2D matrix of pre-assigned cells; 0 for unassigned,
begin
  ϕ = [ ]; -- clauses to be returned.
  S = { }; -- set of pre-assigned cells.
  $X = \{x_{r,c,d} \mid 1 \le r \le n, 1 \le c \le n, 1 \le d \le n\}$;
  for i = 1 to n
    for j = 1 to n
      $\phi = \phi \wedge$ *NaïveExactlyOne*($\{x_{r,c,d} \in X \mid r = i,\ c = j,\ 1 \le d \le n\}$)
        $\wedge$ *NaïveExactlyOne*($\{x_{r,c,d} \in X \mid r = i,\ d = j,\ 1 \le c \le n\}$)
        $\wedge$ *NaïveExactlyOne*($\{x_{r,c,d} \in X \mid c = i,\ d = j,\ 1 \le r \le n\}$)
        $\wedge$ *NaïveExactlyOne*($\{x_{r,c,d} \in X \mid r = br(b,j),\ c = bc(b,j),\ 1 \le b \le n\}$);
      $S = S \cup \{x_{r,c,d} \in X \mid r = i,\ c = j,\ d = M[i][j],\ 1 \le M[i][j] \le n\}$);
    endfor
  endfor
  $\phi = \phi \wedge \bigwedge_{i=1}^{u} S_i$;
  return ϕ;
end

Figure 7. Naïve encoding algorithm for Sudoku

The functions $br(b,j)$ and $bc(b,j)$ return the row and column of the j^{th} cell in a block b.[1] The first four set of clauses refers to each cell, each row, each column and each block having exactly one digit from 1 to n. In addition to these sets, the last one is a set of unit clauses to represent pre-assigned cells; i.e., the pre-assigned cell in row i and column i has a digit from 1 to n; in other words, $1 \le M[i][j] \le n$.

Table 3 shows our experimental results with the naïve encoding on different sized puzzles. It is easy to see that the naive encoding works for small puzzles such as 9×9. Using these encodings for large Sudoku such as 81×81, however, generates huge CNF

[1] The function br and bc can be implemented as follows:

$br(b,j) = (b \,/\, \sqrt{n}) * \sqrt{n} + (j \,/\, \sqrt{n})$

$bc(b,j) = (b \,\%\, \sqrt{n}) * \sqrt{n} + (j \,\%\, \sqrt{n})$

where "/" designates truncating integer division, and "%" designates the modulo operation (remainder after integer division).

files which lead to stack overflow errors during SAT solving. Even the encoding phase was not completed for 100×100 on our machine due to tremendous number of clauses (over 100 million clauses).

	naïve			commander + optimized		
size	variables	clauses	time	variables	clauses	time
9x9	729	11,988	0.04	194	1,263	0.01
16x16	4096	123,904	0.32	551	3,504	0.02
25x25	15625	752,500	1.94	2,794	19,264	0.08
36x36	46,656	3,721,104	11.78	5,170	35,924	0.14
49x49	117,649	11,303,908	33.37	11,555	80,772	0.29
64x64	262,144	33,046,528	105.43	17,256	120,006	0.40
81x81	531,441	85,060,787	overflow	27,021	189,426	0.66
100x100	CNF generation is failed			62,489	439,975	2.26
144x144	CNF generation is failed			58,248	404,061	1.43

Table 3. Experimental results on Sudoku puzzles with different encodings

The runtime of a SAT solver is highly usually unmanagble if the input formula size is on the order of gigabytes. An optimized encoding removes the obvious redundancies when encoding a given problem in CNF. Using an optimized CNF encoding can speed up SAT solving significantly [6,7]. In the case of Sudoku, it is easy to see that a pre-assigned cell implies obvious redundancies in the encodings presented. Since an assigned number at a pre-assigned cell is never changed, all variables representing other numbers associated with a pre-assigned cell can be removed. In addition, variables whose value is duplicated in the same row or column or block are eliminated because duplicated numbers are not allowed within the same region in Sudoku. This optimization idea was given by our previous work [8]. Here, the algorithm is revised to work it in the current context:

Algorithm *Optimize*(X, M) =
begin
  X' = { }; -- set of redundant variables .
  for i = 1 to n
    for j = 1 to n
      $X' = X' \cup \{x_{r,c,d} \in X \mid r = i, c = j, 1 \le d \le n, 1 \le M[i][j] \le n\})$
        $\cup \{x_{r,c,d} \in X \mid r = i, j+1 \le c \le n, d = M[i][j], 1 \le M[i][j] \le n\})$
        $\cup \{x_{r,c,d} \in X \mid i+1 \le r \le n, c = j, d = M[i][j], 1 \le M[i][j] \le n\})$
        $\cup \{x_{r,c,d} \in X \mid r = br(b,j), c = bc(b,j), d = M[i][j], 1 \le M[i][j] \le n$
          $1 \le b \le n\});$
    endfor
  endfor
  return $X - X'$;
end

Figure 8. Optimized algorithm

As shown in Fig. 9, the commander encoding calls the function *Optimize* to eliminate

redundant variables before generating CNF clauses. Compared to the naïve encoding, the set of unit clauses to represent pre-assigned cells exists no longer.

Algorithm *SudokuToCNF*(*M*) =
input
  $M[n][n]$; -- 2D matrix of pre-assigned cells.
begin
  ϕ = [ ]; -- clauses to be returned.
  $X = \{x_{r,c,d} \mid 1 \le r \le n,\ 1 \le c \le n,\ 1 \le d \le n\}$;
  X' = *Optimize*(X, M); -- reduced set of variables.
  for i = 1 to n
    for j = 1 to n
      $\phi = \phi \wedge$ *CmdrExactlyOne*($\{x_{r,c,d} \in X' \mid r = i,\ c = j,\ 1 \le d \le n\}$)
        $\wedge$ *CmdrExactlyOne*($\{x_{r,c,d} \in X' \mid r = i,\ d = j,\ 1 \le c \le n\}$)
        $\wedge$ *CmdrExactlyOne*($\{x_{r,c,d} \in X' \mid c = i,\ d = j,\ 1 \le r \le n\}$)
        $\wedge$ *CmdrExactlyOne*($\{x_{r,c,d} \in X' \mid r = \mathrm{br}(b,j),\ c = bc(b,j),\ 1 \le b \le n\}$);
    endfor
  endfor
  return ϕ;
end

Figure 9. Commander encoding with the optimization

The same puzzles were used to evaluate the commander encoding with the optimization technique. Compared to the naive encoding, the number of variables and clauses is significantly reduced. For instance, the proposed encoding generates 27,021 variables and 189,426 clauses for the 81×81 puzzle; however, the naïve one generates 531,441 variables and 85,060,787 clauses. Since the number of variables and clauses are significantly reduced, MiniSAT solves the puzzle in 0.66 second. Even the largest puzzle (144×144) in our experimental settings was solved in 1.43 seconds.

In summary, the commander encoding with the optimization shows much better performance than the previous naïve encodings used for solving Sudoku with SAT solvers [5,9] in terms of both the number of clauses and SAT solving time. In addition, our combined approach works more efficiently compared to the optimization used alone [8].

5 Related Works

Apart from the pigeon-hole problem and Sudoku puzzle, boolean cardinality selection problems such as “at most one”, “at least one”, and “exactly one” have wide application areas such as production management [10], discrete tomography [11], etc. There has been work done to efficiently encode these constraints in CNF clauses [11,12,13]. In the literature [13], these encodings are compared in terms of 3 criteria (the number of clauses, the number of variables, and decision method).

Compared to the encodings of Warners [12] and $\mathrm{LT}_{\mathrm{PAR}}$ [13], which require searching the state space, our encoding works by unit propagation. Although the encoding of Baileux & Boufkhad [11] requires no search, it generates more clauses

than ours. And the encoding LT_{SEQ} [13] also requires no search and is close to ours in terms of all the 3 criteria above. However, LT_{SEQ} and others mentioned here are devised for "at most many" rather than "at most one". In the problem of "at most one from x_1 through x_{100}", LT_{SEQ} generated 296 clauses, whereas ours generated 294 clauses. Thus, we believe that our technique is better than any others in the problem "select one from a certain set of propositional variables". Apart from the efficiency, our method is flexible so that it can be combined with other techniques to make SAT solvers work fast.

Let us recap our experimental results on the pigeon-hole problem. Our commander encoding with the canonical ordering shows much better performance than the standard naïve encoding for both unsatisfiable cases and satisfiable cases. In Section 3, we used the constraint "each pigeon must be in exactly one hole". It is stronger than the original constraint "each pigeon must be in some hole" used in benchmark test CNF files from SATLIB. We compared our technique with the benchmark test set and our technique resulted in better performance in both the number of clauses and SAT solving time shown in Table 4.

size		SATLIB			commander + ordering		
n	m	vars	clauses	time	vars	clauses	time
5	4	20	45	0.02	29	88	0.01
6	5	30	81	0.02	41	151	0.01
7	6	42	133	0.03	67	248	0.01
8	7	56	204	0.13	101	355	0.02
9	8	72	297	0.87	123	470	0.01
10	9	90	415	9.31	165	610	0.02
11	10	110	561	72.32	215	750	0.02
12	11	132	738	timeout	247	913	0.02
13	12	156	949	timeout	293	1,104	0.02
130	129	16,770	1,081,795	timeout	37,231	142,103	1.42

Table 4. The commander encoding with the canonical ordering shows much better performance than the naïve encoding used in SATLIB benchmark test set.

Recently, various encodings were proposed to formulate Sudoku into a set of CNF clauses [5,9]. The number of clauses generated from these encodings for $n \times n$ Sudoku puzzle is $O(n^4)$. In fact, the extended encoding in [5] is the same as the naïve one described in Section 4. These encodings allow us to solve small instances of Sudoku puzzle such as 9×9 with SAT solvers. However, they generate too many clauses for large size puzzles. This in turn makes the satisfiability checking of the generated CNF clauses difficult. In contrast, the commander encoding with optimization generates very compact CNF, with only $O(n^3)$ clauses, and makes SAT solvers work efficiently. (In Table 3 we observed a reduction of 450X in the size of the CNF encoding for the 81×81 puzzle.)

6 Conclusions

Our contributions can be summarized as follows:

- This paper presents a flexible and efficient encoding for selecting at most one from a certain set of propositional variables. Since our work focuses on the one-object-selection problem, our work can show better performance in the encoding of boolean cardinality constraints such as “at most one” and “exactly one”.
- We demonstrate the effectiveness of our techniques in solving the pigeon-hole problem and Sudoku puzzles. Apart from two applications, we believe that there are many applications needed our techniques such as Hamiltonian path finding [14].

In the future, we plan to implement the LT_{SEQ} encoding and compare it with ours using the same applications mentioned in this paper.

References

1. http://en.wikipedia.org/wiki/Pigeonhole_principle
2. N. Een and N. Sorensson, An Extensible SAT Solver, in The Proceedings of SAT'03, 2003.
3. SAT instances of the Pigeon Hole Problem. (http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/DIMACS/PHOLE/descr.html)
4. http://en.wikipedia.org/wiki/Sudoku
5. I. Lynce and J. Ouaknine, Sudoku as a SAT problem, in The Proceedings of AIMATH’06, 2006.
6. S. Subbarayan and D. Pradhan, NiVER: Non increasing Variable Elimination Resolution for Preprocessing SAT Instances, in The Proceedings of SAT'04, 2004.
7. N. Een and A. Biere, Effective Preprocessing in SAT through Variable and Clause Elimination, in the Proceedings of SAT'05, 2005.
8. G. Kwon and H. Jain, Optimized CNF Encoding for Sudoku Puzzles, in The Proceedings of LPAR’06, 2006.
9. T. Weber, A SAT-based Sudoku Solver, in The Proceedings of LPAR’05, 2005.
10. W. Kuchlin and C. Sinz, Proving Consistency Assertions for Automotive Product Data Management, Journal of Automated Reasoning, Vol.24, pp.45-163, 2000.
11. O. Baileux and Y. Boufkhad, Efficient CNF Encoding of Boolean Cardinality Constraints, in Proceedings of CP 2003, pp.108-122, 2003.
12. J.P. Warners, A Linear-time Transformation of Linear Inequalities into Conjunctive Normal Form, Information Processing Letter, Vol.68, pp.63-69, 1998.
13. C. Sinz, Towards an Optimal CNF Encoding of Boolean Cardinality Constraints, in Proceedings of CP 2005, pp.827-831, 2005.
14. D. Kroning, Hamiltonian Path Finding as SAT, 2005-2006 Lecture on Formal Verification. (From http://people.inf.ethz.ch/daniekro/classes/251-0247-00/ws2005-2006/)