Comparing SAT Encodings of Constraint Problems

Student Name: Mingqian Gao

Student ID: 112122980

Supervisor: Steve Prestwich

Second Reader: Ken Brown

B. Sc. Final Year Project Report

2014

Declaration of Originality.

In signing this declaration, you are confirming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;

- with respect to my own work: none of it has been submitted at any educational institution contributing in any way towards an educational award;

- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Name: Mingqian Gao

Signed: _____

Date: 3rd April 2014

Acknowledgements

A special thank you must go to my supervisor, Dr. Steve Prestwich, for providing me with the opportunity to work on this project that I have great interest, and also for his continued support, guidance and professionalism throughout the project period. I would greatly appreciate all his patience, understanding and inspiration.

Special acknowledgement is due to my second reader Dr. Ken Brown for sparing his precious time to know my project and providing valuable comments and suggestions.

I would also like to thank my parents, my friends, for it is you, who constantly motivates and pushes me to achieve.

Abstract

This project is mainly encoding CSPs (constraint satisfaction problems) by different ways, through SAT (satisfiability) solvers with two different search algorithms, then compare results and study the encodings' performance for each particular problem.

Contents

1 Introduction

1.1 Introduction

In the world, today, CSPs are becoming common, such as many AI problems can be modelled as Constraint Satisfaction Problems (CSPs). In additional, it is well-known that AI is one of techniques are applied in the aspect of processing large-scale data. Therefore, it is fairly worth finding relatively efficient ways to solve them (CSPs). Fortunately, Constraint Satisfaction Problem can be SAT- encoded in more than one way. On the other hand, the choice of encoding is equally important as the selection of search algorithm. Hence, comparing the different encodings in different search algorithms is the main purpose of this project.

1.2 Aim of the Project

This project aims to encode some Constraint Satisfaction Problems and analysis performances under SAT solvers. These are the focus on backtracking search and local search algorithm, which will provide more details in a later section. In this project, three different types of CSPs will be picked up to be encoded, those will have different impact on the performance in the same encoding and same problem will have different impact on performance between encodings.

1.3 Objectives of the Project

This project will include four kinds of encoding and one of these is not published yet. Those are Direct Encoding, Support Encoding, Maximal Encoding and Maximal Support Encoding respectively, based on the advantage and disadvantage of these encodings, performances will be shown by some mathematical approaches. More details on these encodings will appear in a later section.

1.4 Personal Contribution

As described above, it is easy to be seen that CSPs can be solved by some relatively efficient ways but there is not an optimal encoding for every CSP even some CSPs cannot be encoded. The most difficult task is to determine whether there is a model for CSP with finite domains as this is in NP-hard [1]. However, this design is concentrated on the analysis the performance of encodings in search algorithms so that it is not considered the case that no model for CSP with finite domains. Hence, the challenge here would be roughly figuring out the relative optimal in the range of CSPs, which are covered with the CSPs will be discussed in this project.

2 Background

It is necessary to introduce some topics that are required for the following discussion before moving onto further. In this section, from high-level of knowledge on the topic to essential and abstract concepts, would be presented here.

Firstly, Constraint Satisfaction Problems and SAT Encoding will be discussed apparently in this section. After that, SAT solvers and the backtrack search and local search will appear. Eventually, further introduction of SAT Encodings will be presented.

The purpose of above is to give the abstract information for the Analysis & Design and Implementation section. It will be fairly helpful if this section is understood when moving onto next section.

2.1 What is CSPs?

CSPs are the abbreviation of Constraint Satisfaction Problems, which are mathematical problems, defined as a set of objects whose state must satisfy a number of constraints or limitations [2].

Many important problems in AI can be formulated as constraint satisfaction problems (CSPs). A CSP is a triple (V,D,C) where V is a set of variables, D the set of their domains, and C a set of constraints on the variables that specify the permitted combinations of assignments of domain values to subsets of the variables, or equivalently the forbidden combinations. CSPs are usually solved by search algorithms that alternate backtracking with some form of constraint propagation, for example, forward checking or arc consistency

2.2 What challenge is in SAT Encoding?

Encoding problems as Boolean satisfiability (SAT) and solving them with very efficient SAT algorithms has recently caused considerable interest. In particular, local search algorithms have given impressive results on many problems. Unfortunately,

there are several ways of SAT-encoding constraint satisfaction (CSP) and other problems, and few guidelines on how to choose among them. This aspect of problem modelling, like most others, is currently more of an art than a science, yet the choice of encoding can be as important as the selection of search algorithm. It would be very useful to have some guidelines on how to choose an encoding. Experimental comparisons hint at such guideline. [4]

Therefore, it would be easy to encode a CSP to be SAT, while finding an efficient SAT encoding for most even all CSPs is much more difficult. Many journals suggest that SAT encoding have a dramatic impact on runtime of the SAT solver. In additional, the quality of encoding depends on the particular problem, which means there is not a unique benchmark for all problems to assess the quality of encoding. For instance, suppose there are two problems, one is very easy to be solved by direct encoding, however, the other one could be indeterminate by direct encoding but there is probably some encodings have satisfied performance for this problem (It will be shown in the later section).

2.3 What is SAT solver?

SAT problem is the one that determining a given formula whether outputs TRUE for the input of Boolean assignments to each variable. Such as Boolean Circuit, giving series of Boolean input values and determining the output throughout the circuit. Hence, SAT solver is the application that input the Boolean variables and output the result for the SAT problem. As well-known, SAT problem is in NP-Complete, which can be determined in polynomial time rather than solve it. On the other hand, SAT also plays a significant role in many application domains such as AI and Electronic Design Automation (EDA). As a result, SAT solve have received wild range of research attention, and many solver algorithms have been proposed and implemented. The most

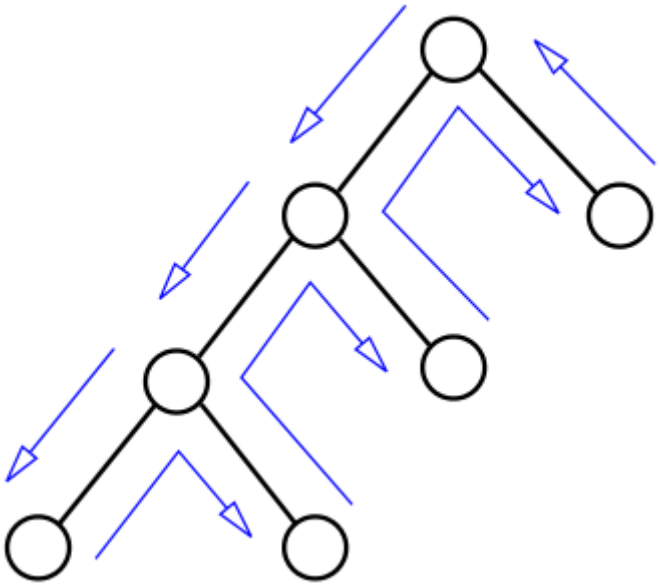
common solver algorithms are backtracking search algorithm and local search algorithm respectively. Following would simply discuss about both solver algorithm.

2.3.1 Algorithms of SAT solvers

1) DPLL algorithm:

“In computer science, the Davis–Putnam–Logemann–Loveland (DPLL) algorithm is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form, i.e. for solving the CNF-SAT problem.” [4]

DPLL is the procedure that owns a powerful efficiency, which has been playing a significant role in most SAT solovers. as well as for many theorem provers for fragments of first-order logic [5]



DPLL Algorithm [5]

As described above, DPLL is completed, therefore, what solvers are applying DPLL algorithm is guaranteed, which means these kinds of solvers are able to determine whether the problem is satisfiable or not. However, the disadvantage of this is that probably will cause over time-consuming. So it is probably a perfect algorithm to compare performances with different encodings rather than computing solutions. In order to settle this embarrassing situation, most researchers are recommending an algorithm called Local search algorithm.

2) Local search algorithm:

"In computer science, local search is a metaheuristic method for solving computationally hard optimization problems. Local search can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions. Local search algorithms move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed." [6]

Since local search, sometimes, would continually search optimal solution in some local, which will lead to bad performance in searching. Therefore, varieties of optimization solution of local search have been coming up, such as the optimal version local search algorithm of Travelling Salesman Problem. However, this project would not give too much detail for above both algorithms because it focuses on comparing different encodings under solvers based on these two algorithms.

2.3.2 SAT solvers

Here would like to present three different SAT solvers, one is based on Local Search algorithm and the others are based on DPLL (one of backtracking search) algorithm.

UBCSAT

"UBCSAT is a Stochastic Local Search SAT Solver framework. It is named after the University of British Columbia, where it was developed by Dave Tompkins under the supervision of Holger Hoos." (see the about page for a longer introduction)[7]. As UBCSAT is based on Local search, it is not guaranteed for the formula (Which means would not result "UNSATISFIABLE"). Therefore, UBCSAT is an uncompleted SAT solver; however, this SAT solver typically would lead a surprising performance in some domains of problem.

MiniSat

MiniSat is a minimalistic implementation solver based on DPLL (a kind of backtracking search) algorithm, which has conflict-clause recording, conflict-driven backjumping, VSIDS dynamic variable order, two-literal watch scheme features [8]

zChaff

zChaff is a SAT solver based on Chaff algorithm. Chaff algorithm is an example of DPLL algorithm, where enhanced performance in an efficient implementation. Many

research suggested, zChaff can run more than one million variables [9], which is greatly good for those encodings that would have a considerable number of variables especially in each clause, such as support encoding in some extreme problems.

.

2.3.3 Usage of solvers

Typically, solvers are required a format to run the model. Hence, many surveys showed two types of format for solvers. However, there is only one format was used, which is CNF format in this project. The other one is called SAT format but here will more focus on CNF format.

CNF format is, simply, described that, 1) There are only three types of the logical operator, namely *AND*, *OR* and *NOT. 2)* Each clause must be composed by disjunction of literals. 3) Clauses must be connected by conjunction.

However, solvers are not required above. Therefore, DIMACS was devised for SAT problem in file format. To input the encoded model, that must be transformed to DIMACS format whatever using CNF format or SAT format. Fortunately, DIMACS format is quiet simple to be transformed from CNF format.

For a DIMACS CNF file, each file, optionally, have single comment lines with the letter “c," for instance, c This is an example of comment line. In the following, a line of the form must be put: **p cnf** *variables clauses*, where variables means the total number of variables in this file; clauses means the total number of clauses in this file.

Then the clauses will be shown by DIMACS style: The way of representation must follow the rules: 1) All variables are described from one(integer number.), that means, supposed, there are three variables then then the format would be 1 2 3 0, where 0 is regards as the end of this clause. 2) Each number in this project has two different types

of representations, which are positive integer and negative integer. Positive one means positive literate and negative is for negative literate.

Based on above rules, it is simple and clearly to create numbers of clauses and those including the header lines to compose a DIMACS CNF file. This is what DIMACS CNF file looks like:

c This is an example

p cnf 3 2

1 2 -3 0

2 -1 0

There are three variables and 2 clauses; variables are represented v_1, v_2, v_3 temporarily in this example. Above is representing that $(v_1 \vee v_2 \vee \neg v_3) \wedge (v_2 \vee \neg v_1)$. Therefore, it is clearly seen that the order of clauses and the order of literals are irrelevant in DIMACS CNF file. [10]

2.4 SAT Encodings.

In this section, 4 different types of SAT encodings will be presented, namely *Direct Encoding*, *Support Encoding*, *Maximal Encoding* and *Maximal Support Encoding*. The first three are published, while *Maximal Support Encoding* is not published yet so that there are quiet limited researched about this encoding can be found on Internet. Therefore, the last encoding would not be given too many details how is working.

2.4.1 Components of Encoding.

It would better to present some basic components of encodings before declaring above four types of encodings. Those components are the set of clauses:

1) At-least-one clauses

As the statement of this clauses, intuitively, these clauses are used for constraining that there is at least one variable in the model. At-least-one is slightly same as *NOT NULL* function in logic programming language. Formally, each CSP variable must be assigned at least one value from a corresponding domain, expressed by [10]:

$$\bigvee_{i} x_{v,i}$$

Where $x_{v,i}$ represents variable x is assigned a value from domain value i.

2) At-most-one clauses

These clauses are a reversed version of At-least-one. However, At-most-one has different with At-least-one in expression. The purpose of At-most-one clauses is to constrain each variable can be allocated no more than one value from domain values. So it is very easy to understand that constrain each assigned-value variable pair to pair, such as [10]:

$$\overline{x_{v,i}} \vee \overline{x_{v,j}}$$

Where i and j are assigned a value, v means domain values.

So far, it would be very easy to understand what above clauses mean, while what following clauses represent would be a little bit abstract.

3) Conflict clauses

What Conflict clauses imply is that it is impossible to assign a value twice in elsewhere. It is not essential for whole variables, but it depends on the particular CSP, such as in Graph Colouring case: assume a given G (V, E, C), where V is the set of vertices, E is set of edges and C represent the set of colours, and let adjacent vertices of the selected vertices have different colour. Here assume only two vertices and two different colour. The clauses is working for this case and the general expression of this type of clauses is:

$$\overline{x_{v,i}} \vee \overline{x_{w,j}}$$

Where v and w are the domains, typically, v $\neq$ w, which means there are two different variables, that is because each variable only have one finite domain of values [10].

4) Support clauses

Support clauses are the reversed version of Conflict clauses. In Conflict clauses, it describes the conflicts from other domains. In contrast, Support clause are declaring the non-conflicts from other domains. However, they are not completely reversed in expression [10]:

$$\overline{x_{w,j}} \vee \left(\bigvee_{i \in S_{v,j,w}} x_{v,i} \right)$$

Where $\neg x_{w,j}$ is the current variable with assigned value j, the literals after that are the set of assigned-value variables that are not conflict with $x_{w,j.}$ Considering if $x_{w,j}$ is chosen but other non-conflicts variables are not chosen, it would lead a FALSE result. This way is probably helpful for understanding what Support clauses describes.

5) Maximality clauses

Maximality clauses are an extending version of Conflict clauses, meanwhile, also are a reversed version of Support clauses. That is because Maximality clauses are representing all other variables that are conflict with current assignment. Conceptually, it is called Maximality clauses since they always force to a maximal number of assignments to each variable. If no assignment to another variables conflicts with a particular assignment such as v = a, then this assignment must be a part of SAT solution [3]. The expression is shown in following [3]:

$$x_{vi} \vee \bigvee_{(w,j) \in K_{vi}} x_{wj}$$

Where $K_{v\,i}$ is the set of pairs (w, j), which makes conflict with $x_{v\,i.}$ In other words, what $x_{w\,j}$ represents is all variables that conflict with current variable $x_{v\,i.}$

2.4.2 Types of Encodings.

The important part of encodings have described in the previous section. Therefore, how construct encodings is the main topic of this section. This project will use four types of encodings, namely *Direct Encoding, Support Encoding, Maximal Encoding and Maximal Support Encoding.*

1) **Direct encoding**

Direct encoding is the most natural, straightforward and widely-use method to encode CSP to SAT. It consists of *At-least-one clauses, At-most-one clauses and Conflicts clauses* [10].

2) **Support Encoding**

Support encoding is an approach that is partial to constrain CSP on non-conflicts. Mostly this encoding is good for CSPs that need not constrain too many conflicts. It consists of *At-least-one clauses, At-most-one clauses and Support clauses* [10].

3) **Maximal Encoding**

The SAT solutions of Maximal encoding are Cartesian products of partial domains such that any combination of values is a solution, and no additional values can be added to the product [10]. It consists of *At-least-one clauses, Conflicts clauses and Maximality clauses* [10].

4) **Maximal Support Encoding**

Maximal Support Encoding is not published yet, but the idea of this encoding is substituting Conflicts clauses to Support clauses. Hence, the whole structure is *At-least-one clauses, Support clauses and Maximality clauses*.

3 Analysis and Design

This section will focus on establishing models for three different Constraint Satisfaction Problems and analyze them.

3.1 Sudoku Problem

3.1.1 Introduction

Sudoku is a logic based, combinational number-placement puzzle. The objective of Sudoku is to fill the $n^2 \times n^2$ grid with the numbers from one to n for each column, each row and each $n \times n$ sub-grid.

Typically, n is equal to 9 but there are many types of Sudoku around the world. In order to reduce the complexity and be able to more focus on analysis, here will only decide to use the standard Sudoku (n = 9). Formally, following will abstract those to be definitions.

Definition 1 *Sudoku is composed by* 9×9 *squared grid, and each Sudoku grid is composed by nine* 3×3 *sub-grids.*

Definition 2 *The objective of Sudoku puzzle is filling numbers from 1 to 9, and each number can be only shown to each row, each column, and each* 3×3 *sub-grid.*

Definition 3 *There is only one solution of each Sudoku puzzle.*

3.1.2 Analisis

It is quiet easy seen that the most challenge on encoding Sudoku is to constrain columns, row, and sub-grids respectively.

The basic and intuitive idea in the solution of Sudoku problem is, each square must exist only one value so that At-least-one clauses and At-most-one clauses must constrain every variable. The challenge of Conflict clauses is to constrain column by

column, row by row and sub-grid by sub-grid. Similarly, Support clauses are also doing the same.

At-least-one: $\bigvee_{i=1}^{n} S_{x,y,i} (1 \leq x, y \leq n)$

Where x and y are the coordinates of Sudoku grid; i is the domain of each square. If the output is based on this formula, it will be like 1, 1, 1, which means square (1, 1) is assigned value 1. In order to represent DIMACS CNF value, there is a transformation from the SAT value. [12]

$$n^2 \times (x - 1) + n \times (y - 1) + i \qquad ①$$

Here is the code for At-least-one clauses：

```
def ALO(n):
    # To ensure each square has at least one value.

    X = range(1, n + 1, 1)
    Y = X
    I = X
    f = open(cnfFlieName, "a")

    for x in X:
        for y in Y:
            for i in I:
                # print >> f, "(%d,%d)%d" % (x, y, i),
                print >> f, "%d" % (n * n * (x - 1) + n * (y - 1) + i),
            print >> f, 0
    return
```

At-most-one: $\neg S_{x,y,i} \vee \neg S_{x,y,j}$

Where $1 \leq x, y \leq n$, $1 \leq i \leq n-1$, $i+1 \leq j \leq n$. Here will also apply the transformation ① to convert SAT values (Same as following SAT values).

```
def AMO(n):
    # No AMO because each domain only has one value.

    X = range(1, n + 1, 1)
    Y = range(1, n + 1, 1)
    I = range(1, n, 1)

    f = open(cnfFlieName, "a")

    for x in X:
        for y in Y:
            for i in I:
                for j in range(i + 1, n + 1, 1):
                    # print >> f, "-%d%d%d -%d%d%d 0" % (x, y, i, x, y, j)
                    print >> f, "-%d -%d 0" % (n * n * (x - 1) + n * (y - 1) + i, n * n * (x - 1) + n * (y - 1) + j)

    return
```

Conflicts-Columns: $\neg S_{x,y,i} \vee \neg S_{m,y,i}$

Where $1 \leq i, y \leq n$, $1 \leq x \leq n-1$, $x+1 \leq m \leq n$.

```
def CON_COLUMN(n):
    X = range(1, n, 1)
    Y = range(1, n + 1, 1)
    I = range(1, n + 1, 1)
    f = open(cnfFlieName, "a")

    for y in Y:
        for i in I:
            for x in X:
                for m in range(x + 1, n + 1, 1):
                    # print >> f, "-(%d,%d)%d -(%d,%d)%d 0" % (x, y, i, m, y, i)
                    print >> f, "-%d -%d 0" % (n * n * (x - 1) + n * (y - 1) + i, n * n * (m - 1) + n * (y - 1) + i)

    return
```

Conflicts-Rows: $\neg S_{x,y,i} \vee \neg S_{x,n,i}$

Where $1 \leq i, x \leq n$, $1 \leq y \leq n-1$, $y+1 \leq m \leq n$.

```
def CON_ROW(n):
    X = range(1, n + 1, 1)
    Y = range(1, n, 1)
    # N = range(y + 1, 9, 1)
    I = range(1, n + 1, 1)
    f = open(cnfFlieName, "a")

    for x in X:
        for i in I:
            for y in Y:
                for n in range(y + 1, n + 1, 1):
                    # print >> f, "-(%d,%d)%d -(%d,%d)%d 0" % (x, y, i, x, n, i)
                    print >> f, "-%d -%d 0" % (n * n * (x - 1) + n * (y - 1) + i, n * n * (x - 1) + n * (n - 1) + i)

    return
```

Conflicts-Sub-grids: $\neg S_{x, y, i} \vee \neg S_{x1, y1, i}$

Where:

$x = k \times (\sqrt{n}) + a$ $y = j \times (\sqrt{n}) + b$

$x_1 = k \times (\sqrt{n}) + a_1$ $y_1 = j \times (\sqrt{n}) + b_1$

$1 \leq a, b, b1 \leq (\sqrt{n})$ $0 \leq k, j \leq (\sqrt{n}) - 1$

$a + 1 \leq a1 \leq (\sqrt{n})$ $1 \leq i \leq n$ $y \neq y_1$

```
def CON_SUBGRIDS(n):
    I = range(1, n + 1, 1)
    Z = range(int(sqrt(n)))
    J = range(int(sqrt(n)))
    X = range(1, int(sqrt(n)) + 1, 1)
    Y = range(1, int(sqrt(n)) + 1, 1)
    L = range(1, int(sqrt(n)) + 1, 1)

    f = open(cnfFlieName, "a")

    for i in I:
        for k in Z:
            for j in J:
                for a in X:
                    for b in Y:
                        for a1 in range(a + 1, int(sqrt(n)) + 1, 1):
                            for b1 in L:
                                if (int(sqrt(n)) * j + b) != (int(sqrt(n)) * j + b1):
                                    # print >> f, "-(%d,%d)%d -(%d,%d)%d 0" % (
                                    #     int(sqrt(n)) * k + a, int(sqrt(n)) * j + b, i, int(sqrt(n)) * k + a1,
                                    #     int(sqrt(n)) * j + b1, i)
                                    print >> f, "-%d -%d 0" % (
                                        n * n * ((int(sqrt(n)) * k + a) - 1) + n * ((int(sqrt(n)) * j + b) - 1) + i,
                                        n * n * ((int(sqrt(n)) * k + a1) - 1) + n * ((int(sqrt(n)) * j + b1) - 1) + i)

    return
```

Support-Columns:

$$\neg S_{z,y,i} \cup \left(\bigvee_{j \in X_{y,x,i,z,y}} S_{y,x,j} \right)$$

Where:

$1 \leq x \leq n$ $1 \leq y \leq n - 1$

$y + 1 \leq z \leq n$ $1 \leq i, j \leq n$

$i \neq j$

```
def SUP_COLUMN(n):
    X = range(1, n, 1)
    I = range(1, n + 1, 1)

    f = open(cnfFlieName, "a")
    num_val = 0

    for x in I:
        for y in X:
            for z in range(y + 1, n + 1, 1):
                for i in I:
                    for j in I:
                        if (i != j):
                            # print >> f, "(%d,%d)%d" % (y, x, j),
                            print >> f, "%d" % (n * n * (y - 1) + n * (x - 1) + j),
                    num_val = num_val + 1
                    # print >> f, "(%d,%d)-%d 0" % (z, x, i)
                    print >> f, "-%d 0" % (n * n * (z - 1) + n * (x - 1) + i)
    return num_val
```

Support-Rows:

$$\neg S_{x,z,i} \cup \left(\bigvee_{j \in X_{x,y,i,x,z}} S_{x,y,j} \right)$$

Where:

$1 \leq x \leq n$ $1 \leq y \leq n - 1$

$y + 1 \leq z \leq n$ $1 \leq i, j \leq n$

$i \neq j$

```
def SUP_ROW(n):
    X = range(1, n, 1)
    I = range(1, n + 1, 1)

    f = open(cnfFlieName, "a")
    num_val = 0

    for x in I:
        for y in X:
            for z in range(y + 1, n + 1, 1):
                for i in I:
                    for j in I:
                        if (i != j):
                            # print >> f, "(%d,%d)%d" % (x, y, j),
                            print >> f, "%d" % (n * n * (x - 1) + n * (y - 1) + j),
                    num_val = num_val + 1
                    # print >> f, "(%d,%d)-%d 0" % (x, z, i)
                    print >> f, "-%d 0" % (n * n * (x - 1) + n * (z - 1) + i)
    return num_val
```

Support-Sub-grids:

$$\neg S_{x,y,i} \cup \left(\bigvee_{j \in Xx1,y1,i,x,y} S_{x1,y1,j} \right)$$

Where:

$x = x_0 + k$ $y = y_0 + l$

$x_1 = x_0 + k_1$ $y_1 = y_0 + l_1$

$1 \leq x_0, y_0 \leq n$ ($x_0, y_0 \bmod \sqrt{n} = 1$)

$0 \leq k, k_1, l, l_1 \leq (\sqrt{n}) - 1$ $1 \leq i, j \leq n$

$i \neq j$ $x \neq x_1$ $y \neq y_1$

```
def SUP_SUBGRIDS(n):
    X = range(1, n + 1, int(sqrt(n)))
    Y = range(1, n + 1, int(sqrt(n)))
    K = range(int(sqrt(n)))
    L = range(int(sqrt(n)))
    I = range(1, n + 1, 1)

    f = open(cnfFlieName, "a")
    num_val = 0


    for x0 in X:
        for y0 in Y:
            for k in K:
                for l in L:
                    for i in I:
                        for k1 in K:
                            for l1 in L:
                                if (k != k1 or l != l1):
                                    for j in I:
                                        if (i != j):
                                            # print >> f, "(%d,%d)%d" % (x0 + k1, y0 + l1, j),
                                            print >> f, "%d" % (n * n * ((x0 + k1) - 1) + n * ((y0 + l1) - 1) + j),
                        num_val = num_val + 1
                        # print >> f, "(%d,%d)-%d 0" % (x0 + k, y0 + l, i)
                        print >> f, "-%d 0" % (n * n * ((x0 + k) - 1) + n * ((y0 + l) - 1) + i)
    return num_val
```

Maximality:

$$S_{x,y,i} \cup \left(\bigvee_{j \in X_{x,y1,i,x,y}} S_{x,y1,j} \right) \cup \left(\bigvee_{j \in X_{x1,y,i,x,y}} S_{x1,y,j} \right) \cup \left(\bigvee_{j \in X_{x3,y3,i,x,y}} S_{x3,y3,j} \right)$$

Where:

$x_3 = x + k$ $y_3 = y + l$ $1 \leq x, x_1, y, y_1, i \leq n$

$1 \leq k, l \leq (\sqrt{n})$ $i = j$

$x \neq x_1$ $y \neq y_1$ $x \neq x_2$ $y \neq y_2$

```
def MAX(n):
    # rows and columns
    X = range(1, n + 1, 1)
    Y = range(1, n + 1, 1)
    I = range(1, n + 1, 1)

    # sub-grids
    K = range(int(sqrt(n)))
    L = range(int(sqrt(n)))

    f = open(cnfFlieName, "a")

    for x in X:
        for y in Y:
            for i in I:
                # To print out row-relevant maximal value
                for y1 in Y:
                    if (y1 != y):
                        # print >> f, "(%d,%d)%d" % (x, y1, i),
                        print >> f, "%d" % (n * n * (x - 1) + n * (y1 - 1) + i),

                # To print out column-relevant maximal value
                for x1 in X:
                    if (x1 != x):
                        # print >> f, "(%d,%d)%d" % (x1, y, i),
                        print >> f, "%d" % (n * n * (x1 - 1) + n * (y - 1) + i),

                # To print out sub-grids-relevant maximal value
                n2 = int((x + sqrt(n) - 1) / sqrt(n))
                m2 = int((y + sqrt(n) - 1) / sqrt(n))
                x2 = Arithmetic_formula(n, n2, int(sqrt(n)))
                y2 = Arithmetic_formula(n, m2, int(sqrt(n)))
                for k in K:
                    if (x2 + k != x):
                        for l in L:
                            if (y2 + l != y):
                                # print >> f, "(%d,%d)%d" % (x2 + k, y2 + l, i),
                                print >> f, "%d" % (n * n * ((x2 + k) - 1) + n * ((y2 + l) - 1) + i),

                # To print out the end of each clause and the corresponding CSP variable
                # print >> f, "(%d,%d)%d 0" % (x, y, i)
                print >> f, "%d 0" % (n * n * (x - 1) + n * (y - 1) + i)

    return
```

In the code of Maximality clauses, there is a helper method that is used for calculating coordinates that sub-grid is located. The basic idea of the part of sub-grids is finding out the location of which sub-grid should be located in, and traversal all squares of this sub-grid to find out the conflict variables.

Based on above clauses, it is very easy to build up the encodings by composing different clauses:

```
def direct_encoding(n):
    outputCNF.write_header(n ** 4, ALO_num + AMO_num + CON_IllegalMove(n), cnfFlieName)

    ALO(n)
    AMO(n)
    CON_IllegalMove(n)


def support_encoding(n):
    outputCNF.write_header(n ** 4, ALO_num + AMO_num + SUP(n), cnfFlieName)

    ALO(n)
    AMO(n)
    SUP(n)


def maximal_encoding(n):
    outputCNF.write_header(n ** 4, ALO_num + CON_IllegalMove(n) + MAX_new(n), cnfFlieName)

    ALO(n)
    CON_IllegalMove(n)
    MAX_new(n)


def maximal_support_encoding(n):
    outputCNF.write_header(n ** 4, ALO_num + CON_IllegalMove(n) + MAX_new(n) + SUP(n), cnfFlieName)

    ALO(n)
    SUP(n)
    MAX_new(n)
```

In additional, the first line of each function is to generate the DIMACS “header”. There is also one thing must be mentioned; since each DIMACS file should put the number of variables and the number of clauses in the first or second line, so calculating the number of those are an essential part during encoding. Fortunately, the number of variables is very easy to figure out. However, the number of clauses, sometimes, is too complicated to calculating before generating the DIMACS CNF file.

This project provides a dynamic tracking for figuring out the number of clauses, the corn idea is when generating clauses, the accumulator will plus one automatically. Hence, the static stuff should be considered before running codes.

The number of variables is easy to calculate. Intuitively, each SAT variable with a value from the domain is variable of DIMACS CNF. Hence, in Sudoku example, there are 81 squares and each square have a nine-value's domain. In other words, the number of variables must be $81 \times 9 = 729$. Replacing 9 to n, the formula is the number of variables = n^3.

The number of clauses is more challenge than that of variables, and that is because some clauses is difficult enough to calculate. Fortunately, there are still a significant number of cases are able to calculate, and those cases that are too hard to calculate can be done by dynamic tracking as the description.

The number of At-least-one: n^2

The number of At-most-one: $n^3 \times (n - 1) / 2$

The number of Conflicts: $n^3 \times (n - 1) + (n \times (n - 1) / 2 - n \times ((\sqrt{n}) - 1)) \times n^2$

The number of Support: Dynamic tracking because of the complication.

The number of Maximality: n^3

The experiments will be shown in the next chapter; there is only analysis and encoding to be shown in this chapter.

3.2 Knight's Tour Problem

3.2.1 Introduction

Knight's Tour Problem is an instance of Hamiltonian Path Problem and is required to visit every square only once on the chessboard based on a knight. There are two different type; one is closed, which means if the knight can return to the origin after visiting every square; the other one is open, which means if the knight is no way to back to the origin after visiting every square. Additionally, the exact number of open case on an 8×8 chessboard is still unknown even if Knight's Tour problem can be solved in linear time. [13][16]

There are two interesting conclusions are proved by Schwenk and Cull *et al.* & Conrad *et al.* respectively:

1) *For any m × n board with m ≤ n, a closed knight's tour is always possible unless one or more of these three conditions are met* [14]*:*

   m and n are both odd

   m = 1, 2, or 4

   m = 3 and n = 4, 6, or 8.

2) *On any rectangular board whose smaller dimension is at least 5, there is a (possibly open) knight's tour*[15][16]

3.2.2 Definitions

Abstracting the information to be definitions is great helpful for modeling. Therefore, abstracting is an essential step on encoding.

Chess is well-known around the world so that it is easy to understand moves of a knight. First of all, considering the chessboard as coordinates as Sudoku problem discussed above. The range of the legal move is in x-coordinate ± 2 and y-coordinate; meanwhile the absolute value of range-values must not be equal. That is the most challenge for Knight's Tour problem when building up the model.

Definition 1 Knight's Tour problem is a sequence of move on a given chessboard, based on the principle of knight in chess. The legal move is the range of following:

Suppose (x, y) is current square, all square of legal-move is

(x $\pm$ i, y $\pm$ j) where i, j $\in$ {-2, -1, 1, 2} and |i| $\neq$ |j|.

Definition 2 Every time in a square can be represented by a timestamp. When the value of the last timestamp = the number of square on the chessboard, which means visiting is finished.

3.2.2 Analysis

According to the definitions above, it is straightforward to build up a relational tuple(x, y, t), where x and y are the current locations; t is the timestamp on current location. Furthermore, the set of t is the domain for each square.

At-least-one:

$$\bigvee_{t=1}^{n^2} S_{x,y,t}$$

Where $1 \leq x, y \leq n$; n is the size of bound(Here focus on the square board)

As this type variables are not working in DIMACS CNF file so that there is another transformation for Knight's Tour problem:

$$n^3 \times (x - 1) + n^2 \times (y - 1) + i$$

Because the size ofthedomain is n^2 so that coordinates need plus one in exponent. On the other hand, there is a difference between parameters t and i or j in codes. Keeping in mind all i or j in codes represent timestamp as well.

```
def ALO(n):
    # To ensure each square has at least one value.
    f = open(cnfFlieName, "a")

    X = range(1, n + 1, 1)
    Y = range(1, n + 1, 1)
    I = range(1, n * n + 1, 1)

    for x in X:
        for y in X:
            for i in I:
                print >> f, "%d" % (n * n * n) * (x - 1) + (n * n) * (y - 1) + i),
            print >> f, 0
    return
```

At-most-one: $\neg S_{x, y, i} \vee \neg S_{x, y, j}$

Where

$1 \leq x, y \leq n$ $1 \leq i \leq n^2$ $i + 1 \leq j \leq n^2$.

```
def AMO(n):
    # To ensure each square has at most one value.

    f = open(cnfFlieName, "a")

    X = range(1, n + 1, 1)
    Y = range(1, n + 1, 1)
    I = range(1, n * n + 1, 1)

    for x in X:
        for y in Y:
            for i in I:
                for j in range(i + 1, n * n + 1, 1):
                    print >> f, "-%d -%d 0" % (
                        (n * n * n) * (x - 1) + (n * n) * (y - 1) + i, (n * n * n) * (x - 1) + (n * n) * (y - 1) + j)
                    # print >> f, "-(%d,%d)%d -(%d,%d)%d 0" % (x, y, i, x, y, j)
    return
```

Conflicts:

The Conflicts clauses are a kind of complicated because there in only several legal-move squares in whatever how large the board is. In other word, if n is linear increasing then the complexity of these clauses will appear a $4 \times n^3$ of growth!

In additional, there are two helper methods that are used for detecting the current location is in legal-move square or not.

```
# To judge current coordinate is in legal squares set or not.
def In_LegalSquare(legal_square, x_current, y_current):
    if ((x_current, y_current) in legal_square):
        return True
    return False


# To get the legal squares set.
def Legal_Squares(x_origin, y_origin):
    legal_move = [-2, -1, 1, 2]
    legal_square = []

    for a in legal_move:
        for b in legal_move:
            if ((fabs(a) != fabs(b))):
                # At most 16 squares to be searched
                legal_square.insert(len(legal_square), (x_origin + a, y_origin + b))
    # print legal_square
    return legal_square
```

In order to cover all the conflicts, there are three different Conflicts clauses.

x and y represent the current location, i and j mean the timestamp of current location.

$$1 \le x, y \le n \qquad 1 \le i \le n^2 - 1 \qquad 1 \le j \le n^2$$

1) $\neg S_{x,y,i} \cup \neg S_{x1,y1,i}$ where x1 and y1 is the location of the next move;

2) $\neg S_{x,y,i} \cup \neg S_{x1,y1,i+1}$ $1 \le x1, y1 \le n$ and x1,y1 is not in the range of legal-move squares. *(work for 1) and 2))*

3) $\neg S_{x,y,i} \cup \neg S_{x1,y1,j}$ where x1 and y1 is the location of the next move; $1 \le x1, y1 \le n$ and x1,y1 is in the range of legal-move squares. $j \neq i + 1$

The code of the picture is in following page:

```
def CON_IllegalMove(n):
    # Use 2 helper method
    # Legal_Squares()
    # In_LegalSquare

    f = open(cnfFlieName, "a")

    legal_move = [-2, -1, 1, 2]
    X = range(1, n + 1, 1)
    Y = range(1, n + 1, 1)
    I = range(1, n * n, 1)
    J = range(1, n * n + 1, 1)


    # legal_square = []
    num_cla = 0

    for x in X:
        for y in Y:
            # reduce computing times.
            legal_square = Legal_Squares(x, y)
            for x1 in X:
                for y1 in Y:
                    if ( (x1 > 0) and (x1 <= n) and (y1 > 0) and (y1 <= n) and ((x != x1) or (y != y1))):
                        if not (In_LegalSquare(legal_square, x1, y1)):
                            for i in I:
                                num_cla = num_cla + 1
                                print >> f, "-%d -%d 0" % ((n * n * n ) * (x - 1) + (n * n) * (y - 1) + i,
                                                           (n * n * n ) * (x1 - 1) + (n * n) * (y1 - 1) + i)
                                # print >> f, "-(%d,%d)%d -(%d,%d)%d" % (x, y, i, x1, y1, i)
                                #
                                num_cla = num_cla + 1
                                print >> f, "-%d -%d 0" % ((n * n * n ) * (x - 1) + (n * n) * (y - 1) + i,
                                                           (n * n * n ) * (x1 - 1) + (n * n) * (y1 - 1) + i + 1)
                                # print >> f, "-(%d,%d)%d -(%d,%d)%d" % (x, y, i, x1, y1, i + 1)
                        else:
                            for i in I:
                                for j in J:
                                    if (j != i + 1):
                                        num_cla = num_cla + 1
                                        print >> f, "-%d -%d 0" % ((n * n * n ) * (x - 1) + (n * n) * (y - 1) + i,
                                                                   (n * n * n ) * (x1 - 1) + (n * n) * (y1 - 1) + j)
                                        # print >> f, "-(%d,%d)%d -(%d,%d)%d" % (x, y, i, x1, y1, j)

    return num_cla
```

Support:

These clauses are perfect for this problem, because how large is the bound value is not relevant to the size of legal-move squares. Therefore, the view and the encoding is quiet simple.

$$\neg S_{x,y,i} \cup \left(\bigvee_{(x1,y1) \in X_{x',y'}} S_{x1,y1,i+1} \right)$$

Where $X_{x',\,y'}$ is the set of locations of legal-move squares. $1 \leq i \leq n^2 - 1$

```
def SUP(n):
    f = open(cnfFlieName, "a")

    legal_move = [-2, -1, 1, 2]
    X = range(1, n + 1, 1)
    Y = range(1, n + 1, 1)
    I = range(1, n * n, 1)

    num_cla = 0

    for x in X:
        for y in Y:
            # reduce computing times.
            legal_square = Legal_Squares(x, y)
            for i in I:
                for x1 in range(x - 2, x + 3, 1):
                    for y1 in range(y - 2, y + 3, 1):
                        if ( (x1 > 0) and (x1 <= n) and (y1 > 0) and (y1 <= n) and ((x != x1) or (y != y1))):
                            if (In_LegalSquare(legal_square, x1, y1)):
                                print >> f, "%d" % ((n * n * n ) * (x1 - 1) + (n * n) * (y1 - 1) + i + 1),
                                # print >> f, "(%d,%d)%d" % (x1, y1, i + 1),
                num_cla = num_cla + 1
                print >> f, "-%d 0" % ((n * n * n ) * (x - 1) + (n * n) * (y - 1) + i)
                # print >> f, "-(%d,%d)%d 0" % (x, y, i)

    return num_cla
```

Maximality:

Maximality clauses will be more complicated related to Support clauses, because there is the reversed work need to be done. In order to find out all conflicts, it is necessary to split two parts. One is in the legal range; the other one is not in the legal range.

$$S_{x,y,i} \cup \left(\bigvee_{(x1,y1) \in X_{x',y'}} S_{x1,y1,j1} \right) \cup \left(\bigvee_{(x2,y2) \in \neg X_{x',y'}} S_{x2,y2,j2} \right)$$

Where:

x, y is the location of the current square. i is the timestamp of current location.

$X_{x',y'}$ is the set of locations of legal-move squares. In other words, x1 and y1 is in legal range rather than x2, y2

$1 \leq i \leq n^2$

$i \leq j1 \leq i + 1$

$1 \leq j \leq n^2$, where $j \neq i$

```
def MAX_new(n):
    f = open(cnfFlieName, "a")

    legal_move = [-2, -1, 1, 2]
    X = range(1, n + 1, 1)
    Y = range(1, n + 1, 1)
    I = range(1, n * n + 1, 1)

    num_cla = 0

    for x in X:
        for y in Y:
            # reduce computing times.
            legal_square = Legal_Squares(x, y)
            for i in I:
                for x1 in X:
                    for y1 in Y:
                        if ( (x1 > 0) and (x1 <= n) and (y1 > 0) and (y1 <= n) and ((x != x1) or (y != y1))):
                            if not (In_LegalSquare(legal_square, x1, y1)):
                                for j in range(i, i + 2, 1):
                                    if j <= n**2:
                                        print >> f, "%d" % ((n * n * n) * (x1 - 1) + (n * n) * (y1 - 1) + j),
                                        # print >> f, "(%d,%d)%d" % (x1, y1, j),
                            else:
                                for j in I:
                                    if (j != i + 1):
                                        print >> f, "%d" % ((n * n * n) * (x1 - 1) + (n * n) * (y1 - 1) + j),
                                        # print >> f, "(%d,%d)%d" % (x1, y1, j),
                num_cla = num_cla + 1
                print >> f, "%d 0" % ((n * n * n) * (x - 1) + (n * n) * (y - 1) + i)
                # print >> f, "(%d,%d)%d 0" % (x, y, i)
    return num_cla
```

The encodings is in following:

```
def direct_encoding(n):
    outputCNF.write_header(n ** 4, ALO_num + AMO_num + CON_IllegalMove(n), cnfFlieName)

    ALO(n)
    AMO(n)
    CON_IllegalMove(n)


def support_encoding(n):
    outputCNF.write_header(n ** 4, ALO_num + AMO_num + SUP(n), cnfFlieName)

    ALO(n)
    AMO(n)
    SUP(n)


def maximal_encoding(n):
    outputCNF.write_header(n ** 4, ALO_num + CON_IllegalMove(n) + MAX_new(n), cnfFlieName)

    ALO(n)
    CON_IllegalMove(n)
    MAX_new(n)


def maximal_support_encoding(n):
    outputCNF.write_header(n ** 4, ALO_num + CON_IllegalMove(n) + MAX_new(n) + SUP(n), cnfFlieName)

    ALO(n)
    SUP(n)
    MAX_new(n)
```

3.3 N - Queens Problem

N – Queens problem describe that placing n chess queens in an n×n chessboard then a solution of N – Queens is to let n queens will not attack each other. Generally, a classic version is Eight – Queens but the challenage is to find the number of solutions for a particular value n. There is still no known formula to figure out how many solutions exist. Howevery, Watkins, John J. had proved that there is at least on solution when n = 1 and n $\geqslant$ 4. [17]

How does a queen attack other chess? Each row, each column and each diagonal of the position of the queen is possible to be the objective of the attack.

3.3.1 Definitions

Definition 1 Placing n queens into a n×n chessboard and protect them from attack.

Definition 2 Each row, column and diagonal of the place of the queen can be the objective of attack. The aim is to prevent each queen sharing the same row, column and diagonal.

3.3.2 Analysis

The most straightforward method is checking each row; column and diagonal of the position of each queen to see if they share a way in the given chessboard. However, a better way is to prune the redundant constraint.

Considering aspects of row, column and diagonal. Rows always do the same work as columns' so that reduce works of rows or columns is the same. Assume a given chessboard Q(X, Y , I), where X and Y are the sets of coordinates; I is the set of BOOLEAN value. Simpley, constraining (1, 1, T) from (1, 2, F) to (1, 4, F) and after that, turning to (1,2, T). The problem is from (1, 1, F) or (1, 3, F)? Suppose, it is from (1, 1, F), which will have two Conflict clauses: $\neg$(1, 1, T) $\vee$ $\neg(1, 2, F)$ and $\neg$(1, 1, F) $\vee$ $\neg(1, 2, T)$. They are the same clause, because both clauses are representing only one variable is chosen. The reduction is working in this way. Similarly, this

approach is also working for diagonals. Therefore, the better approach of constraint is just comparing the rear variables.

In terms of BOOLEAN, 1 means TRUE, and there is a queen in this square, vise versa.

At-least-one:

$$\bigvee_{i=0}^{1} S_{x,y,i}$$

Where $1 \leq x, y \leq n$

```
def ALO(n):
    # To ensure each square has at least one value.

    X = range(1, n + 1, 1)
    I = [0, 1]
    f = open(cnfFlieName, "a")

    for x in X:
        for y in X:
            for i in I:
                # print >> f, "(%d,%d)%d" % (x, y, i),
                print >> f, "%d" % (2 * n * (x - 1) + 2 * (y - 1) + i + 1),
            print >> f, 0
    return
```

At-most-one: $\neg S_{x,y,1} \cup \neg S_{x,y,0}$

Where $1 \leq x, y \leq n$

```
def AMO(n):
    # No AMO because each domain only has one value.

    X = range(1, n + 1, 1)
    Y = range(1, n + 1, 1)
    i, j = 1, 0

    f = open(cnfFlieName, "a")

    for x in X:
        for y in Y:
            # print >> f, "-%d%d%d -%d%d%d 0" % (x, y, i, x, y, j)
            print >> f, "-%d -%d 0" % (2 * n * (x - 1) + 2 * (y - 1) + i + 1, 2 * n * (x - 1) + 2 * (y - 1) + j + 1)

    return
```

Conflicts：

$$\neg S_{x,y,1} \cup \neg S_{x1,y1,0}$$

Where $1 \leq x, y \leq n$, $x1,y1 \in$ the coordinate set of rows, columns and diagonals of the current square.

```
def CON_COLUMN(n):
    X = range(1, n, 1)
    Y = range(1, n + 1, 1)
    I = [0, 1]
    f = open(cnfFlieName, "a")

    num_cla = 0

    for y in Y:
        for x in X:
            for m in range(x + 1, n + 1, 1):
                # print >> f, "-(%d,%d)%d -(%d,%d)%d 0" % (x, y, 1, m, y, 0)
                num_cla = num_cla + 1
                print >> f, "-%d -%d 0" % (2 * n * (x - 1) + 2 * (y - 1) + 1 + 1, 2 * n * (m - 1) + 2 * (y - 1) + 0 + 1)

    return num_cla
```

```
def CON_ROW(n):
    X = range(1, n + 1, 1)
    Y = range(1, n, 1)
    I = [0, 1]
    f = open(cnfFlieName, "a")

    num_cla = 0

    for x in X:
        for y in Y:
            for n in range(y + 1, n + 1, 1):
                # print >> f, "-(%d,%d)%d -(%d,%d)%d 0" % (x, y, 1, x, n, 0)
                num_cla = num_cla + 1
                print >> f, "-%d -%d 0" % (2 * n * (x - 1) + 2 * (y - 1) + 1 + 1, 2 * n * (x - 1) + 2 * (n - 1) + 0 + 1)

    return num_cla
```

```
def CON_DIAGONALS(n):
    X = range(1, n, 1)
    Y = range(1, n, 1)

    I = [0, 1]

    f = open(cnfFlieName, "a")
    num_cla = 0

    for x in X:
        for y in Y:
            for x1 in range(x + 1, n + 1, 1):
                for y1 in range(y + 1, n + 1, 1):
                    if (fabs(x - x1) == fabs(y - y1)):
                        # print >> f, "-(%d,%d)%d -(%d,%d)%d 0" % (x, y, 1, x1, y1, 0)
                        num_cla = num_cla + 1
                        print >> f, "-%d -%d 0" % (
                            2 * n * (x - 1) + 2 * (y - 1) + 1 + 1, 2 * n * (x1 - 1) + 2 * (y1 - 1) + 0 + 1)

    return num_cla
```

Support:

$$\neg S_{x,y,1} \cup \left(\bigvee_{(x1,y1) \in X_{x',y'}} S_{x1,y1,1} \right)$$

Where $1 \leq x, y \leq n$, $x1,y1 \in$ the coordinate set of non-rows, non-columns and non-diagonals of the current square.

```
def SUP(n):
    X = range(1, n, 1)
    Y = range(1, n, 1)

    I = [0, 1]

    f = open(cnfFlieName, "a")
    num_cla = 0

    for x in X:
        for y in Y:
            for x1 in range(x + 1, n + 1, 1):
                for y1 in range(y + 1, n + 1, 1):
                    if ((fabs(x - x1) != fabs(y - y1)) and (x != x1) and (y != y1)):
                        # print >> f, "(%d,%d)%d" % (x1, y1, 1),
            # print >> f, "-(%d,%d)%d 0" % (x, y, 1)
            #
                        print >> f, "%d" % (
                            2 * n * (x1 - 1) + 2 * (y1 - 1) + 1 + 1),
            num_cla = num_cla + 1
            print >> f, "-%d 0" % (
                2 * n * (x - 1) + 2 * (y - 1) + 1 + 1)

    return num_cla
```

Maximality:

$$S_{x,y,1} \cup \left(\bigvee_{(x1,y1) \in X_{x',y'}} S_{x1,y1,0} \right)$$

Where $1 \leq x, y \leq n$, $x1,y1 \in$ the coordinate set of rows, columns and diagonals of the current square.

```
def MAX(n):
    X = range(1, n, 1)
    Y = range(1, n, 1)

    I = [0, 1]

    f = open(cnfFlieName, "a")
    num_cla = 0

    for x in X:
        for y in Y:
            for x1 in range(x, n + 1, 1):
                for y1 in range(y, n + 1, 1):
                    if ((fabs(x - x1) == fabs(y - y1)) or (x == x1) or (y == y1)):
                        if not (x == x1 and y == y1):
                            # print >> f, "(%d,%d)%d" % (x1, y1, 0),
            # print >> f, "(%d,%d)%d 0" % (x, y, 1)
            #
                        print >> f, "%d" % (
                            2 * n * (x1 - 1) + 2 * (y1 - 1) + 0 + 1),
            num_cla = num_cla + 1
            print >> f, "-%d 0" % (
                2 * n * (x - 1) + 2 * (y - 1) + 1 + 1)

    return num_cla
```

The encodings are consist of above clauses, such as:

```
#
# Encodings in following

def direct_encoding(n):
    outputCNF.write_header(ALO_val, ALO_cla + AMO_num + CON_COLUMN(n) + CON_ROW(n) + CON_DIAGONALS(n), cnfFlieName)

    ALO(n)
    AMO(n)
    CON_COLUMN(n)
    CON_ROW(n)
    CON_DIAGONALS(n)


def support_encoding(n):
    outputCNF.write_header(ALO_val, ALO_cla + AMO_num + SUP(n), cnfFlieName)

    ALO(n)
    AMO(n)
    SUP(n)



def maximal_encoding(n):
    outputCNF.write_header(ALO_val, ALO_cla + CON_COLUMN(n) + CON_ROW(n) + CON_DIAGONALS(n) + MAX(n), cnfFlieName)

    ALO(n)
    CON_COLUMN(n)
    CON_ROW(n)
    CON_DIAGONALS(n)
    MAX(n)


def maximal_support_encoding(n):
    outputCNF.write_header(ALO_val, ALO_cla + SUP(n) + MAX(n), cnfFlieName)

    ALO(n)
    SUP(n)
    MAX(n)
```

As the study of these three problems, the following section will implement the models; propose hypothesises and prove hypothesises by analyzing data.

4 Implementation

All the preparations are done by long pages. In this section, the main tasks are implementing the models created above. Then, it will give two different comparisons; one is the same problem, different encodings; the other one is same encoding, different problem.

There are some hypothesizes to be proposed:

1) Less clauses better performance?
2) Local search has better performance than DPLL's
3) Add symmetry clauses can make Local search easier,but backtracking will be harder.
4) Direct encoding shows better performance because it has less size of literals in a clause.

4.1 Experiments

In the experiments, it will mainly use Control Variable Method, which is making only one variable into experiments.

Firstly, the three problems will be appeared the result respectively to study the performances of encodings.

4.1.1 Sudoku Problem

Direct Encoding			
Params	Sudoku		
	n = 9	n = 16	n = 25
Number of variables	729	4096	15625
Number of clauses	10287	110848	688125
UBCsat time	0 s	0.02 s	N/A
MiniSat time	0.01 s	0.220013 s	0.912057 s
SATISFIABLE			

Support Encoding			
Params	Sudoku		
	n = 9	n = 16	n = 25
Number of variables	729	4096	15625
Number of clauses	9558	96512	578750
UBCsat time	0.09 s	0.46 s	4.40 s
MiniSat time	0.056003 s	98.0981 s	3608.19 s
SATISFIABLE			INDETERMINATE

Maximal Encoding			
Params	Sudoku		
	n = 9	n = 16	n = 25
Number of variables	729	4096	15625
Number of clauses	8100	84224	516250
UBCsat time	0 s	0.01 s	0.190 s
MiniSat time	0.012 s	0.552034 s	6.62441 s
SATISFIABLE			

Maximal Support Encoding			
Params	Sudoku		
	n = 4	n = 9	n = 16
Number of variables	729	4096	15625
Number of clauses	7371	69888	406875
UBCsat time	0	0	0
MiniSat time	0.044002 s	1.52809 s	99.2662 s
SATISFIABLE			

Above four tables are showing the running time from UBCSat (Local Search based) and MiniSat (Backtracking Search based), it is very clear seen, the runtime from UBCsat always have a shorter time. There is an interesting place in direct encoding

table, where n = 25, the runtime of UBCsat is N/A that is because when the model was run by UBCsat, it came up errors about its interior problem. It might also because Local Search is uncompleted and unstable so that lead to the errors. Comparing Direct Encoding table with Maximal Encoding table, it suggests that Maximal Encoding not always show the best performance when the problem has limited numbers of conflicts clauses. However, looking Support Encoding table, it shows the worst performance on Sudoku example. That is the result of lots of Support clauses. As the mentions in the previous chapter, Support clauses are focus on non-conflicts, and when the problem performs the simple non-conflicts constraint, it will be much better than what it is showing in this case. Additionally, Support clauses sometimes have very large size of literals in one clause, which will cause the solver spend great deal of time on interpreting one clause. By contrast, Maximal Support Encoding is not published yet but it shows a steady performance in this case even if when n become larger, runtime shows the incredible growth.

Overall, in this case, UBCsat shows an unstable performance but it also shows off the significant searching speed in some encodings such as Support Encoding and Maximal Encoding. It is surprising Direct Encoding shows the best performance in Sudoku example because Sudoku example has tons of Support clauses, and that means, conceptually, it perfectly fits Maximal encoding. On the other hand, the number of clauses is not relevant to performance between encodings. However, it cannot eliminate the possibility that Sudoku is a special case so that more analysis is necessary.

4.1.1.2 Knight's Tour Problem

Params	Knight's Tour		
	n = 4	n = 6	n = 8
Number of variables	256	1296	4097
Number of clauses	18496	295716	1803440
UBCsat time	0 s	0.01 s	0.190 s
MiniSat time	0.032002 s	0.424026 s	295.966 s
SATISFIABLE			

Support Encoding			
Params	Knight's Tour		
	n = 10	n = 6	n = 8
Number of variables	10000	1296	4096
Number of clauses	505000	23976	133120
UBCsat time	0.130 s	0.00 s	0.020 s
MiniSat time	0.992062 s	0.036002 s	0.156009 s
SATISFIABLE			

Maximal Encoding			
Params	Knight's Tour		
	n = 4	n = 6	n = 8
Number of variables	256	1296	4097
Number of clauses	16832	274332	1803440
UBCsat time	0 s	0.02 s	0.220 s
CPU time	0.072004 s	8.97256 s	291.734 s
SATISFIABLE			

Maximal Support Encoding			
Params	Knight's Tour		
	n = 10	n = 6	n = 8
Number of variables	10000	1296	4097
Number of clauses	20000	2592	8192
UBCsat time	0 s	0.00 s	0.0 s
MiniSat time	142.865 s	0.80405 s	13.5448 s
SATISFIABLE			

Looking Support Encoding table, it is not surprising that Support encoding gets the best performance in this case, because Knight's Tour is a problem that own just few legal moves from the previous position; that means non-conflict clauses are very limited and simply find them out. In contrast, it will be very harsh for Maximal encoding because in Maximal encoding, it aims to find out all conflicts. However, Direct encoding shows a stable performance when n is not too large. Furthermore, the number of clauses is not relevant to performance again be proved in this case.

4.1.1.3 N - Queens Problem

Direct Encoding			
Params	N-Queens		
	n = 32	n = 64	n = 128
Number of variables	2048	8192	32768
Number of clauses	44208	351584	2804416
UBCsat time	N/A	N/A	N/A
MiniSat time	0.024001 s	0.33202 s	8.12051 s
SATISFIABLE			

Maximal Encoding			
Params	N-Queens		
	n = 32	n = 64	n = 128
Number of variables	2048	8192	32768
Number of clauses	44145	351457	2804161
UBCsat time	0.010 s	0.480 s	7.720 s
MiniSat time	0.056003 s	0.708044 s	15.549 s
SATISFIABLE			

Support Encoding			
Params	N-Queens		
	n = 16	n = 32	n = 64
Number of variables	512	2048	8192
Number of clauses	736	3008	12160
UBCsat time	N/A	N/A	N/A
MiniSat time	0.036002 s	42.4147 s	183.287 s
SATISFIABLE			

Maximal Support Encoding			
Params	N-Queens		
	n = 16	n = 32	n = 64
Number of variables	512	2048	8192
Number of clauses	662	2854	11846
UBCsat time	0 s	0.00 s	0.010 s
MiniSat time	0.036002 s	35.4702 s	179.007 s
SATISFIABLE			

Similarly, Direct encoding and Maximal encoding show the grand performance in this case that is because N-Queens problem also has lots of conflict clauses. Furthermore, looking at Maximal Encoding table, the growth of MiniSat is lower than that of UBCsat; it proves that adding symmetry breaking clauses can promote the performance of backtracking solves instead of local search solves. [4, 18, 19]

4.1 Conclusion

Overall, for those hypothesizes, it is no doubt that the number of clauses is not relevant to performance between encodings, but they are more depended on the conflicts or non-conflicts clauses. Maximal encoding and Support encoding are two different extreme approach of modeling a problem from aspects of conflicts.

Second one is difficult to say because they all have owned advantages. Local search is more concentrated on efficiency but not guaranteed, while DPLL more focus on completeness. Therefore, different people, different views.

For the third one, although it did not appear a straightforward way to prove it, but in the analysis, add symmetry breaking clauses can make backtracking easier not only was mentioned in some researches [4, 18, 19], but also giving an instance of this thesis in above study.

In terms of Direct encoding, there are still lots more arguments in this project because the amount of data is not enough prove something. However, Direct encoding still shows related firm in those fluctuant examples it might because of the number of literals.

Experimental comparisons of this project are quite limited,and there still need to work on to complete the proof. In the future, it will constrain more CSPs with more encodings; comparing the relationship between encodings, between similar problems. It is possible to prove NP is equal to P or not by analyzing the subtle relationships.

Bibliography

[1] David Poole and Alan Mackworth. (April 2010). Possible Worlds, Variables, and Constraints. In: Artificial Intelligence: Foundations of Computational Agents. University of British Columbia

[2] Wikipedia. (26 March 2014). *Constraint satisfaction problem.* Available: http://en.wikipedia.org/wiki/Constraint_satisfaction_problem. Last accessed 31 March 2014.

[3] Steven Prestwich. (2004). In Proceedings of CP-2004. Full dynamic substitutability by SAT encoding. p512--526.

[4] Steven Prestwich. (2003). *Local Search on SAT-Encoded Colouring Problems.* Lecture Notes in Computer Science. p105—119.

[5] Wikipedia. (2014). DPLL algorithm. Available: http://en.wikipedia.org/wiki/DPLL_algorithm. Last accessed 31 March 2014.

[6] Wikipedia. (2014). Local_search_(optimization). Available: http://en.wikipedia.org/wiki/Local_search_(optimization). Last accessed 31 March 2014.

[7] Dave Tompkins. (2012). UBCSAT. Available: http://ubcsat.dtompkins.com/home. Last accessed 08 March 2014

[8] Niklas Eén, Niklas Sörensson. The MiniSat page. Available: http://minisat.se/Main.html. Last accessed 08 March 2014.

[9] zChaff Solver Homepage. Available: https://www.princeton.edu/~chaff/zchaff.html. Last accessed 08 March 2014.

[10] Steve Prestwich. (2009). CNF Encodings. In: Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh Handbook of satisfiability. Amsterdam; Washington, DC: IOS Press. p75-p97.

[11] Ines Lynce and Joel Ouaknine. (2006). Sudoku as a SAT Problem. *Proceedings of AIMATH*

[12] Steve Prestwich, *A Symbolic Approach to SAT Encoding*

[13] Wikipedia. (2014). Knight's tour. Available: http://en.wikipedia.org/wiki/Knight's_tour. Last accessed 08 March 2014.

[14] Allen J. Schwenk (1991). "Which Rectangular Chessboards Have a Knight's Tour?". Mathematics Magazine: 325–332.

[15] Cull, P.; De Curtins, J. (1978). "Knight's Tour Revisited". Fibonacci Quarterly 16: 276–285.

[16] Conrad, A.; Hindrichs, T.; Morsy, H. & Wegener, I. (1994). "Solution of the Knight's Hamiltonian Path Problem on Chessboards". Discrete Applied Mathematics 50 (2): 125–134. doi:10.1016/0166-218X(92)00170-Q.

[17] Watkins, John J. (2004). Across the Board: The Mathematics of Chess Problems. Princeton: Princeton University Press. ISBN 0-691-11503-6

[18] S. D. Prestwich. Negative Effects of Modeling Techniques on Search Performance. Annals of Operations Research vol. 18, Kluwer Academic Publishers, 2003, pp. 137–150.

[19] S. D. Prestwich, S. Bressan. A SAT Approach to Query Optimization in Mediator Systems. Fifth International Symposium on the Theory and Applications of Satisfiability Testing, University of Cincinnati, 2002, pp. 252–259. Submitted to Annals of Mathematics and Artificial Intelligence.

[20] Nieuwenhuis, Robert; Oliveras, Albert; Tinelly, Cesar (2004), "Abstract DPLL and Abstract DPLL Modulo Theories", Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2004, Proceedings: 36–50