Reduction of Sudoku Puzzle to Exact Cover and Implementing the Exact Cover Equivalent Using Dancing Links

A THESIS PROPOSAL

Presented to the Faculty of
School of Computer Studies
MSU - Iligan Institute of Technology
Iligan City

In Partial Fulfillment
of the Requirements for the Degree
BACHELOR OF SCIENCE IN COMPUTER SCIENCE

Josiah Eleazar Regencia
Kristel Ahlaine Gem Pabillaran
June 2017

Abstract

The Sudoku puzzle has been well-known to many for several years and studies have come up with various algorithms to solve the Sudoku problem directly. Most studies present algorithms with good theoretical running time but solutions are not optimal. The Brute-force search is a known optimal solution but at the expense of a very slow running time depending on the difficulty of the Sudoku problem. As shown in (Takayuki Yato, 2003) the Sudoku is NP-Complete, it can then be shown that the Sudoku problem can be reduced into another NP-Complete problem, specifically, the Exact Cover problem. While the Exact Cover can be solved using the same brute-force approach, Knuth's Dancing Links implementation provided optimization which promised a faster practical running time (Knuth, 2000).

In 2014, Harrysson described and implemented the reduction of human-playable Sudokus with clues ranging from 17 to 31 and observed that the while it is faster to reduce Sudoku puzzles with more clues, the difficulty of finding the solution does not reside in the number of clues, but on something else (Mattias Harrysson, 2014). The motivation of this paper is to describe and implement the reduction of puzzles that were not covered by (Mattias Harrysson, 2014) : the non-human playable Sudokus - puzzles with less than 17 clues and with multiple solutions. The process will involve reduction of sudoku to DLX. Runtime of the reduction and of solving the solutions will be observed and will be compared with the general observations of Harryson. Experiments will be conducted to selected suduko puzzles with 0 up to 16 clues.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Background of the Study

Sudoku is a logic-based, combinatorial number puzzle, where the objective is to fill a 9×9 grid with digits so that each column, row, and block contains all of the digits from 1 to 9. In 2006, it was found out that the classic 9×9 Sudoku has 6,670,903,753,021,072,936,960 or about 6.67×10^{21} valid Sudoku solutions (Bertram Felgenhauer, 2006). For a Sudoku with 16×16 grid, the number of solutions is not known (Kapanowski, 2010). Sudoku grids with less than 17 clues have the potential to have more than one solution (Gary McGuire, 2012). Thus, a valid human-playable Sudoku must have at least 17 clues. These human-playable puzzles can be solved using pencil and paper.

	2			3		9		7
	1							
4		7				2		8
		5	2				9	
			1	8		7		
	4				3			
				6			7	1
	7							
9		3		2		6		5

6	2	8	5	3	4	9	1	7
5	1	9	8	7	2	4	3	6
4	3	7	9	1	6	2	5	8
8	6	5	2	4	7	1	9	3
3	9	2	1	8	5	7	6	4
7	4	1	6	9	3	5	8	2
2	5	4	3	6	9	8	7	1
1	7	6	4	5	8	3	2	9
9	8	3	7	2	1	6	4	5

Figure 1.1: Sudoku Puzzle

Solving Sudoku puzzles are both challenging and difficult. In Computer Science term, the Sudoku problem is in the complexity class of NP-complete problems (Takayuki Yato, 2003). The Sudoku problem finds a solution wherein for each row, column and box, there are no repeated digits. To solve the problem just like all other NP-complete problems, the straightforward approach to Sudoku problems is to employ a brute-force algorithm with backtracking. This is relatively slow but the solution is optimal. (Patrik Berggren, 2012) also compared backtracking with rule-based and Boltzman Machines but only for individual solutions of the 17-clue puzzle.

There are alternative methods for this. Faster approaches for NP-complete algorithms are Approximations and Randomized Algorithm, which include Genetic Algorithms, Tabu Search, and Simulated Annealing. As a consequence, the optimal solutions are not guaranteed in these alternative approaches.

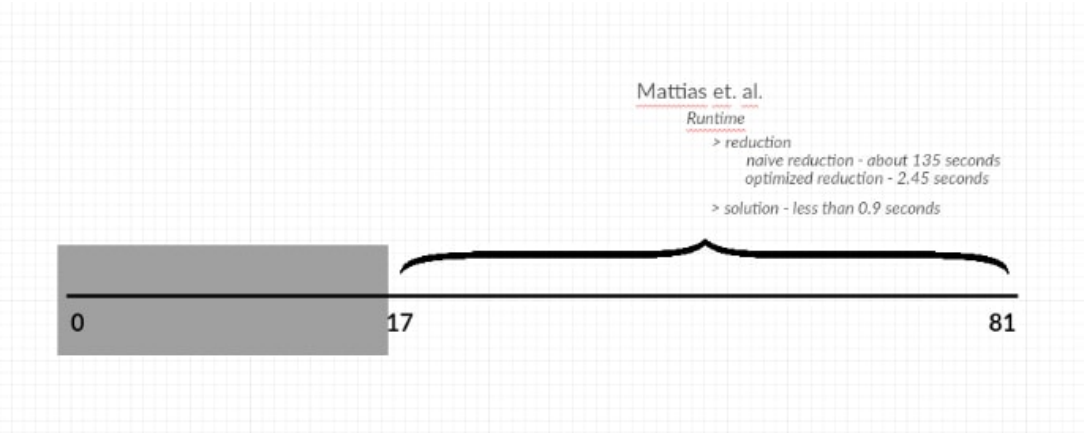
Sudoku solutions can be tackled by reducing Sudoku problems into instances of Exact Cover (Mattias Harrysson, 2014). In Computer Science, this is called Reduction. Any NP-Complete problem can be reduced in a polynomial time to another instance of NP-Complete problem. Once reduced, the Sudoku problem can be solved by solving its equivalent Exact Cover instance.

The Exact Cover finds a set of subsets wherein the intersection of the subsets is empty and the union of the subsets is the major set. Knuths Algorithm X is a known algorithm in finding such subsets in an Exact Cover. Algorithm X is a recursive, non deterministic, depth-first, backtracking algorithm. (Knuth, 2000) referred to Algorithm X as the most obvious trial-and-error approach for finding all solutions to the exact cover problem, which is represented by the Algorithm X using a matrix A consisting of 0s and 1s.

Also, a notable optimization of Algorithm X is Knuth's Dancing Links or DLX. Knuth observed that a naive implementation of his Algorithm X would spend an inordinate amount of time, O(n) searching for 1s. In DLX, only the 1s are considered and are implemented using doubly-linked list. Because Exact Cover problems tend to be sparse, this representation is usually much more efficient in both size and computational time (Dahlke, 2008). In 2014, Mostrales, in her undergraduate thesis, presented a java implementation of the DLX, but was not anchored into any application. Same approach will be used in implementing DLX. This time, however, it will be implemented for Sudoku instances. It would also be interesting to verify if DLX can find the 6.67×10^{21} solutions of Sudoku within a considerable time limit.

When solving the Sudoku through its Exact Cover equivalent, the difficulty does not reside in the number of clues, but on something else (Mattias Harrysson, 2014). Exploring human-playable Sudoku puzzles (17 or more clues), the reduction of (Mattias Harrysson, 2014) shows

promising results in solving the Sudoku in its Exact Cover equivalent. However, it remains to show how the reduction process can be scaled for multi-solution Sudoku puzzles and whether if it also affects the runtime. **Add Results of Mattias Harrysson**



(a) Solving 0 - 16 clue Sudoku

Figure 1.2

Statement of the Problem

This study intends to describe and implement non-human playable Sudoku puzzles (less than 17 clues) based on the reduction optimization by (Mattias Harrysson, 2014) and also whether claim of Harrysson that the difficulty in solving Sudoku in its Exact Cover equivalent is not based on the number of clues also applies to non-human playable Sudoku puzzles.

Objectives of the Study

The aim of this study is to reduce non-human playable Sudoku puzzles to its Exact Cover equivalent and implementing the DLX algorithm. Specifically:

1. As basis for comparison, to implement a usual brute force solution for Sudoku.
2. To solve non-human playable Sudoku puzzles through:
   (a) Describing the reduction to Knuths DLX as shown in (Mattias Harrysson, 2014).
   (b) Implementing the DLX algorithm.
   (c) Experimenting different Sudoku puzzles.

3. Compare the results with the findings in (Mattias Harrysson, 2014).

Limitations

This study will focus on continuing the study of Harrysson but with the focus on exploring non-human playable Sudoku puzzles; which were not explored in the study of Harrysson At the same time, this study will be limited on solving 9×9 Sudoku problems and not its other variants. This study will also only emphasize on the reduction and on solving the Exact Cover equivalent using DLX.

Significance

According to (Mattias Harrysson, 2014), the reduction process is arguably the hardest part in the study since there is not much detailed literature about the reduction process. By solving non-human playable Sudoku puzzles in its Exact Cover equivalent, this study supplements previous studies that tackle solving Sudoku through reduction. Also, this study provides a description on how the reduction process of non-human playabe Sudoku puzzles to its Exact Cover equivalent is done. This will also hope to spark the interest of other software engineers to take advantage of reduction as a mechanism to solve real-world problems.

Chapter 2

Literature Review

Many practical algorithms used for solving the Sudoku problems are fast enough to solve the puzzle but the solutions are not optimal. Thus, in solving the Sudoku problem, the optimal way of doing it would be through brute-force search and the most known algorithm for the brute-force search is backtracking (Cazenave, 2012). However, this solution is very slow and inefficient. (Knuth, 2000) introduced a practically faster technique by applying a doubly-linked list to an algorithm he discovered, Algorithm X. The technique is called the Dancing Links (DLX). Knuth said that The Algorithm X and DLX, however, can only be applied to an Exact Cover Problem. As shown by (Takayuki Yato, 2003), the Sudoku problem is NP-Complete by constructing an $n^2 \times n^2$ instance of Sudoku from an $n \times n$ instance of a Latin Square Completion. Thus being an NP-Complete problem, the Sudoku problem is reducible to an Exact Cover problem, which is also an NP-Complete problem.

(Patrik Berggren, 2012) compared three algorithms for solving the sudoku problem with the primary focus to measure and analyze the potential of these three algorithms in solving 17-clue Sudoku problems consisting of 49, 151 puzzles with a maximum solving time limit of 20 seconds. The algorithms were backtracking, rule-based and Boltzmann Machine. In addition, they also covered the difficulty rating of a sudoku puzzle, sudoku puzzle generation and how well the algorithms are fit for parallelizing. Out of the 49, 151 17-clue puzzles, the backtracking algorithm was not able to solve 1, 150 puzzles; 142 of those unsolved puzzles had unstable measurements. However, the algorithm was the second most efficient algorithm among the three tested algorithms. It had a mean solving time of 1.66 seconds for the solved puzzles. Given the maximum time limit 20 seconds to solve each puzzle, (Patrik Berggren, 2012) suggests that if there had been more time, the other 1,150 unsolved puzzles would have been solved. However, the solving time for each puzzle may be different from each other and the solving times are unknown, it is impossible to know what their solving time is. Solving all 49, 151 17-clue puzzles with no unstable measurements, the Rule-based algorithm ended being the most efficient and accurate solving algorithm with a solving time of 0.02 seconds. With the use of logic rules with a polynomial time complexity, the thesis observed that there was

a small time interval at which puzzles were solved. However, when the algorithm resolves to guessing, the time complexity is changed to exponential time thus the solving time will increase substantially - because, as noted, the guessing is simply like the backtracking algorithm. Using Boltzmann Machines proved to be very inefficient in solving the Sudoku problem. The algorithm was not able to solve all 49, 151 17-clue puzzles. Instead it needed at least 46 clues for it to be able to solve a puzzle. All results were still given a maximum solving time limit of 20 seconds. But the results turned out to be terrible and the measurements were unstable. In the end, as shown, only the backtracking algorithm and the rule-based algorithm were found to be capable of solving the sudoku problem.

(Mattias Harrysson, 2014) reduced 9×9 human-playable Sudoku puzzles (17 or more clues) to an exact cover problem in the form of a matrix containing of only 1s and 0s. Each rule in sudoku is considered a constraint; thus, having four constraints: cell constraint, row constraint, column constraint and box constraint. The solution must cover all columns. Each cell in grid G has nine candidates which means that there must be nine rows in M for each cell where the first cell has 1s in the first column for the first nine rows, the second cell has 1s in the second column and so forth (Mattias Harrysson, 2014). Since there are 81 cells, there must be 81 columns for the cell constraint. And since each requires nine rows, M now therefore has 729 rows. At the same time, each constraint require 81 columns. Since there are four constraints, M would have a total of 324 columns. Hence, when reduced to an exact cover problem, a 9×9 sudoku can be represented as a 729×324 matrix of 1s and 0s. However, instead of using the original binary matrix, (Mattias Harrysson, 2014) optimized its reduction by creating the links directly, thus reducing it directly to its Dancing Links form. This optimized reduction produced more efficient results than the naive reductions. The approximate runtime for the naive reductions was around 135 seconds while the approximate runtime for the optimized reductions was 2.54 seconds. (Mattias Harrysson, 2014) only used Sudoku grids with 17 to 31 clues. As input, Harrysson and Laestander used all 49,151 17-clue Sudoku puzzles and gathered 14,236 Sudoku grids with clues from 18 to 31; majority of those grid having around 21 to 31 clues. Each grid timed from the start of the reduction to when a solution was found. Using the optimized reduction, results showed that the solving time of most Sudoku problems were solved in less than 0.5 seconds. All solutions, however, were solved in less than 0.9 seconds. In fact, most Sudoku grids were solved below 0.1 seconds. However, no experiment was made on finding

all 6.67×10^{21} valid Sudoku solutions.

The Pentomino puzzle is an example of an exact cover problem. Backtracking and depth-first traversal are examples of approaching the pentomino puzzle (Nivasch, 2004). As an example, Nivasch used an 8×8 square with a 2×2 hole in the middle and constructed a matrix with 72 columns - 60 for the cells on the board and 12 for the pentominoes. The first algorithm works by selecting the columns sequentially from the first 12 while the second algorithm works by selecting columns sequentially from the other 60. Nivasch implemented a program wherein an arbitrary 60-cell pattern is accepted as an input. The program then proceeds to build the matrix corresponding to the pattern being input, and applies the exact cover algorithm printing all the solutions it finds. The 8×8 square mentioned above, found 520 solutions.

The N Queens problem is another exact cover reducible problem (Knuth, 2000). There are many algorithms for solving the n-queens problem. This includes Genetic Algorithm (GA), Simulated Annealing (SA), Backtracking (BT), and Brute-force (BF) algorithms (Mukherjee, Datta, Chanda, & Pathak, 2015). They proposed an algorithm which is the proposed genetic algorithm and the proposed simulated annealing using genetic algorithm and compared them together with brute-force and backtracking algorithms and found out that the Proposed Simulated Annealing using Genetic Algorithm takes less time than the Proposed Genetic Algorithm as it works on only one solution at a time and processes it to get the result but Proposed Genetic Algorithm gives better solution than the proposed Simulated Annealing using Genetic Algorithm as it works on a population of solutions and processes them to get the result. For Brute-force and backtracking, these algorithms can provide exact results but cannot provide solutions in ample time when the number of N is increased. Therefore, these two algorithms are not efficient to solve N-queens problem whereas the other two mentioned which is the proposed genetic algorithm and proposed simulated annealing using genetic algorithm provide good solutions to higher valued N.

At the same time, in the field of computational music analysis, (Brian Bemman, 2015) reduced one of Milton Babbitts works, the Sheer Pluck, to an exact cover problem to find a sequence of distinct, non-overlapping aggregate regions that completely and exactly covers an irregular matrix of pitch class integers. The Sheer Pluck was constructed into a six-part

all-partition array. The result of the array was a projection consisting of 6 rows and 696 pitch classes. The paper used a backtracking algorithm which yielded a sequence of 58 distinct partitions and aggregate-forming integer compositions that exactly cover the 696 class pitch integers.

While some of the above-mentioned papers have stated that the transformation from Sudoku to an Exact Cover is part of their paper, it is rather difficult to find a complete mapping of the Sudoku to exact cover, especially on Sudoku puzzles with clues. Further, while it was mentioned that the DLX implementation of Sudoku is much faster than the Brute Force in the exact cover equivalent, it is worth investigating the actual run time through our expirements.

Chapter 3

Theoretical Framework

The computational complexity theory was the primary basis why the indirect solution of solving Sudoku using DLX is possible. According to computational complexity theory, computational problems are grouped together according to their inherent difficulty. Many complexity classes are defined using reduction. It is therefore worthy to discuss complexity classes in order to appreciate the reduction method.

This section is organized to contain the following theoretical foundations that are relevant in understanding reduction:

- Complexity classes
- Reduction

Complexity Classes

There are about 70 known complexity classes (Sanjeev Arora, 2007). Probably the most studied classes P, NP, NP-Complete and NP-hard.

Nondeterministic Polynomial (NP) is a set of decision problems solvable in polynomial time by a theoretical non-deterministic Turing machine. Informally, it is a set of decision problems for which the instances where the answer is yes have efficiently verifiable proofs and these proofs have to be verifiable by a deterministic computation that can be performed in polynomial time. A language L is in NP if and only if there exists a polynomial p and q deterministic Turing machine M, such that:

1. For all x and y, the machine M runs in time $p(|\ x\ |)$ on input (x, y),
2. For all x in L, there exists a string y of length $q(|\ x\ |)$ such that $M(x, y) = 1$,
3. For all x not in L and all strings y of length $q(|\ x\ |)$, $M(x, y) = 0$.

On the other hand, Polynomial (P) is a generalization of NP. It contains all decision problems that can be solved by a deterministic Turing machine in polynomial time. Equivalently, it

is the class of decision problems where each yes instance has a polynomial size certificate, and certificates can be checked by a polynomial time deterministic Turing machine. Cobhams thesis says that P is the class of computational problems that are efficiently solved or tractable. Formally, a language L is in P if and only if there exists a deterministic Turing machine M, such that:

1. M runs for polynomial time on all inputs,
2. for all x in L, M outputs 1 and
3. for all x not in L, M outputs 0.

Furthermore, a decision problem C is NP-complete if

1. C is in NP,
2. Every problem in NP is reducible to C in polynomial time, C can be shown to be in NP by demonstrating that a candidate solution to C can be verified in polynomial time. The most notable characteristic of NP-complete problems is that no fast solution to them is known. The time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows.

Lastly, NP-Hard is a class of every problems where every problem L in NP can be reducible in polynomial time to a certain problem H; thus we can say H is NP-Hard. For a certain NP-Complete to be considered a NP-Hard problem, there has to be a polynomial-time reduction from the NP-Complete problem to the NP-Hard problem. Basically, NP-Hard class of problems are as hard as the hardest problems in NP.

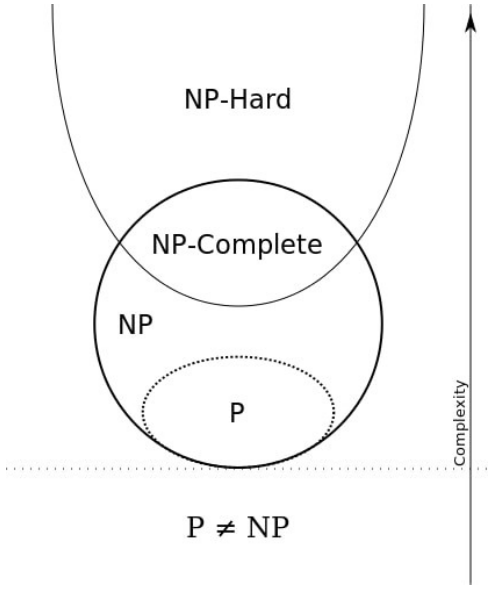


Figure 3.1: Venn Diagram for Class of Complexity Problems

Sudoku problem and Exact Cover problems are known to be NP-Complete.

The decision question of the Sudoku problem is:

"Is there a solution such that there is no repeated digit in each row, column, and box?"

The decision question of the Exact Cover problem is:

"Is there a set of subsets such that the intersection of the subsets is empty and the union of the subsets is the set?"

Exact Cover

An exact cover is when each element in a given set X is contained in exactly one subset of X such that the union of all subsets of a given set X covers the set X and the intersection of all subsets of the set X is empty.

Let S = {A, B, C, D, E, F} be a collection of subsets of a set X = {1, 2, 3 ,4, 5, 6, 7} such that:

- A = {1, 4, 7};
- B = {1, 4};
- C = {4, 5, 7};
- D = {3, 5, 6};
- E = {2, 3, 6, 7}; and
- F = {2, 7}

Alternatively, the above S can also be represented as a binary matrix as below

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	1	0	0	1	0	0	1
B	**1**	0	0	**1**	0	0	1
C	0	0	0	1	1	0	1
D	0	0	**1**	0	**1**	**1**	0
E	0	1	1	0	0	1	1
F	0	**1**	0	0	0	0	**1**

Then the subcollection S* = {B, D, F} is an exact cover since:

- B ∪ D ∪ F = X; and
- B ∩ D ∩ F = Ø

Formally, given a collection S of subsets of X, an exact cover of X is the subcollection S* of S that satisfies the following conditions:

a. The intersection of any two distinct subsets in S* is empty (the subsets in S* are pairwise disjoint)

b. The union of the subsets in S* is X.

Reduction

NP-complete problems are studied because the ability to quickly verify solutions to a problem (NP) seems to correlate with the ability to quickly solve that problem (P). It is not known whether every problem in NP can be quickly solvedthis is called the P versus NP problem. But if *any NP-complete problem* can be solved quickly, then *every problem in NP* can, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every NP-complete problem (that is, it can be reduced in polynomial time). Because of this, it is often said that NP-complete problems are *every problem in NP* or *more difficult* than NP problems in general.

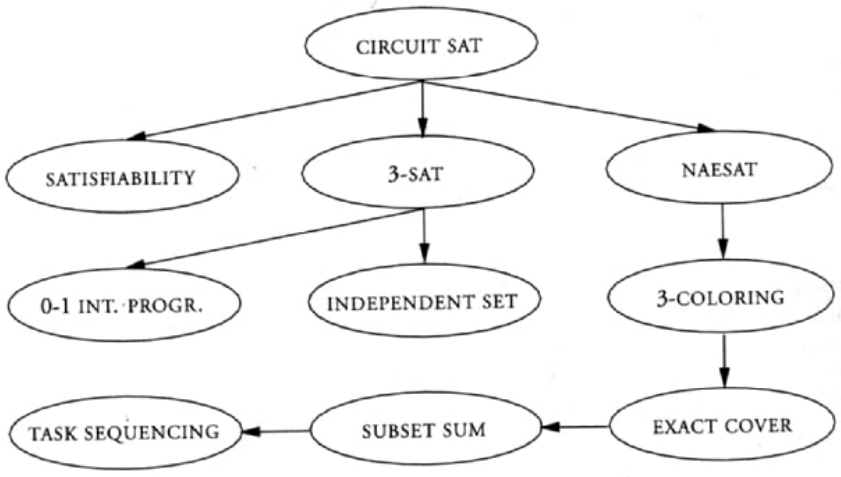


Figure 3.2: Reduction of CircuitSat Problem to other known NP problems

Since any NP-complete problem can be reduced to another NP-Complete problem, its solution can be derived indirectly using reduction.

Reduction is an algorithm transforming one problem into another problem such that, under N. If a language A $\subseteq$ {0, 1}* is polynomial-time Karp reducible to a language B $\subseteq$ {0, 1}* denoted by A $\leq$ p B if there is a polynomial-time computable function f : {0, 1}* $\rightarrow$ {0, 1}* such that for every x $\in$ {0, 1}*, x $\in$ A if and only if f(x) $\in$ B.

The figure above illustrates the reduction of other CircuitSat Problem to other known NP problems. The reduction of CircuitSat to 3-Sat has led to the growth of the number of problems belonging to NP-Complete class.

In this study, the reduction of Sudoku to Exact Cover will have two implications:

- That the solution of Sudoku in can be solved through the terms of Exact Cover.
- That the theoretical runtime of solving through exact cover is still the within the same class.

Chapter 4

Methodology

The brute force method will be used as the basis of comparison of the other solutions. Such that the Methodology is organized as follows:

1. Brute-force Implementation
2. Reduction of Sudoku to Exact Cover
3. Solving the Exact Cover through DLX

Lastly, the Sudoku puzzles will be classified so that the algorithms performance can be observed.

For uniform notation, the following is defined:

- Each row in the Sudoku grid will be denoted as r.
- Each column in the Sudoku grid will be denoted as c.
- Each value in a cell in the Sudoku grid will be denoted as n.
- The Sudoku puzzle will be denoted as P.
- All possibilities or combinations in the Sudoku puzzle is denoted as a three-tuple $P_i = \{ r, c, n \}$.
- The Exact Cover X will be a four-tuple $X_j = \{ cellCons, rowCons, colCons, boxCons\}$ such that it is the st of given clues in a Sudoku problem.

Brute-force Implementation

Analyzing the results of the Brute-force implementation for Sudoku will show the difference in efficiency in solving the Sudoku problem. Generally, when using the Brute-force algorithm in solving any problem, one has to implement its four main procedures: *first*, *next*, *valid*, and *output*. These procedures should take a data P as a parameter for the particular instance of the problem to be solved. The procedures are also defined as the following:

1. first (P): generate a first candidate solution for P.

2. next (P, n): generate the next candidate P after the current one n.

3. valid (P, n): check whether candidate n is a solution for P.

4. output (P, n): use the solution n of P as appropriate to the application.

The *first* procedure should return *null* if there are no candidates of the instance P. The *next* procedure must also tell that when there are no more candidates for the instance P, after the current one n. The Brute-force algorithm is the expressed as

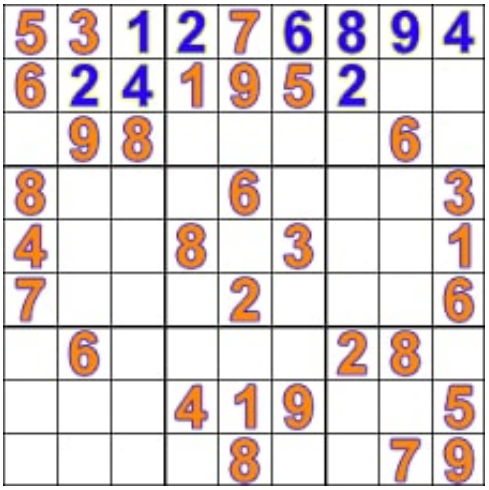
Algorithm 1 Brute-force Algorithm

```
1: n ← first(P)
2: while n ≠ null
3:     if valid(P, n) then output(P, n)
4:     n ← (P, n)
5: end while
```

In implementing the Brute-force algorithm in the Sudoku problem, the algorithm will visit empty cells in some order defined by the programmer. When visiting each cell, the algorithm fills in digits in sequence of available choices and backtracks once a dead-end is reached. For every backtrack, the algorithm iterates the digit in the cell most recently filled before the the cell where the dead end was seen. If all possible digits are already tried by the algorithm, the algorithm again does a backtrack and repeats the same process until a solution is found. The algorithm of the backtracking will be as follows:

Algorithm 2 Backtracking Algorithm

```
1: if choices(empty) then true
2: for 1 → 9
3:     try choice n
4:     if solved(n) then true
5:     unmake choice n
6: return false
```



Brute-force: orange digits are fixed while blue ones are partial solutions

Reduction of Blank Sudoku to Exact Cover

Finding a more efficient way of solving the Sudoku problem is one of the main focuses of this paper, the next step will be the implementing of the reduction of the Sudoku problem to an exact cover problem. In the standard 9×9 Sudoku variant, in which each of 9×9 cells is assigned one of 9 numbers, there are 9×9×9=729 possibilities. Using obvious notation for rows, columns and numbers, the possibilities can be labeled R_1C_1**#1**, R_1C_1**#2**, ..., R_9C_9**#9** (Harryson and Laestander, 2014). (Harryson and Laestander, 2014) provides detail in reducing the Sudoku problem to an exact cover problem containing four constraints:

a. cell constraint
b. row constraint
c. column constraint
d. box constraint

The fact that each kind of constraint involves exactly one of something is what makes Sudoku an exact hitting set problem. The constraints can be represented by constraint sets. The problem is to select possibilities such that each constraint set contains (i.e., is hit by) exactly one selected possibility. (Mattias Harrysson, 2014) uses the following convention for Cell Constraint rows, Row Constraint rows, Column Constraint rows, and Box Constraint rows.

To avoid confusion in the format, the matrix will be arranged as cell constraint, row constraint, column constraint, and then box constraint. Such that the first 81 columns of the matrix will belong to the cell constraint and the next 81 columns will belong to the row constraint as is

shown in the representation below.

The following possibilities in the Sudoku Grid should be transformed to its Exact Cover form as shown below:

$P_i = \{ 1, 1, 1 \} \Rightarrow X_j = \{ 1, 82, 163, 244 \}$

$P_i = \{ 1, 1, 2 \} \Rightarrow X_j = \{ 1, 83, 164, 245 \}$

...

$P_i = \{ 1, 1, 1 \} \Rightarrow X_j = \{ 81, 162, 243, 324 \}$

Or as shown in the illustration below:

	Constraints															
$P_i = \{ r, c, n \}$	**cell**				**row**				**column**				**box**			
	1	**2**	**...**	**81**	**82**	**83**	**...**	**162**	**163**	**164**	**...**	**243**	**244**	**245**	**...**	**324**
$P_i = \{ 1, 1, 1 \}$	1	0	...	0	1	0	...	0	1	0	...	0	1	0	...	0
$P_i = \{ 1, 1, 2 \}$	1	0	...	0	0	1	...	0	0	1	...	0	0	1	...	0
...								...								
$P_i = \{ 9, 9, 9 \}$	0	0	...	1	0	0	...	1	0	0	...	1	0	0	...	1

Cell Constraint: Each cell in the Sudoku Grid has nine candidates which means that there must be nine rows in the Exact Cover Matrix for each cell where the first cell has 1s in the first column for the first nine rows, the second cell has 1s in the second column and so forth. For example, the constraint set for row 1 and column 1, which can be labeled r1c1, contains the 9 possibilities for row 1 and column 1 but different numbers.

$r_1c_1 = \{r_1c_1\textbf{\#1}, r_1c_1\textbf{\#2}, r_1c_1\textbf{\#3}, r_1c_1\textbf{\#4}, r_1c_1\textbf{\#5}, r_1c_1\textbf{\#6}, r_1c_1\textbf{\#7}, r_1c_1\textbf{\#8}, r_1c_1\textbf{\#9}\}$.

Row Constraint: In each cell of the row constraint, the 1s are placed in a new column for each row of the nine rows in the Exact Cover Matrix. The same pattern is applied for the first nine cells in the Sudoku Grid - first 81 rows in the Exact Cover Matrix. The next row constraint starts at the 10th cell of the Sudoku Grid or at the 82nd row of the Exact Cover Matrix. And the next constraint starts at the 19th cell or at the 163rd row and so on and so forth.

For example, the constraint set for all columns of row 1 having the value 1, which can be labeled r1 **#1**, contains the 9 possibilities for row 1 and number 1 but different columns:
r_1**#1** = {r_1c_1**#1**, r_1c_2**#1**, r_1c_3**#1**, r_1c_4**#1**, r_1c_5**#1**, r_1c_6**#1**, r_1c_7**#1**, r_1c_8**#1**, r_1c_9**#1**}.

Column Constraint: The pattern in the column constraint looks similar to the row constraint pattern where the 1s are placed in a new column for the first nine rows in the Exact Cover Matrix. This goes on until the ninth cell. However, unlike the row constraint, the next column constraint starts 1 cell forward from where the last column constraint ended.

For example, the constraint set for all rows in column 1 having the value 1, labeled c1 **#1**, contains the 9 possibilities for column 1 and number 1 but different rows:
c_1**#1** = {r_1c_1**#1**, r_2c_1**#1**, r_3c_1**#1**, r_4c_1**#1**, r_5c_1**#1**, r_6c_1**#1**, r_7c_1**#1**, r_8c_1**#1**, r_9c_1**#1**}.

Box Constraint: Unlike the previous constraints, the box constraint would seem to an irregular pattern at first. For the constraint of three complete boxes, if they are in the same box in the Sudoku Grid, their columns are equal in the Exact Cover Matrix. Since each box has similar rules with the row and column, that each box must only have unique integer from one to nine, in each cell in the Sudoku Grid the columns placed just like the row constraint in the Exact Cover Matrix and continues until the next cell. For example, the constraint set for box 1 (top-most left-most box) having the value 1 in each cell, which can be labeled B1 **#1**, contains the 9 possibilities for the cells in box 1 and number 1:
B_1**#1** = {r_1c_1**#1**, r_1c_2**#1**, r_1c_3**#1**, r_2c_1**#1**, r_2c_2**#1**, r_2c_3**#1**, r_3c_1**#1**, r_3c_2**#1**, r_3c_3**#1**}.

Since there are 9 rows, 9 columns, 9 boxes and 9 numbers, there are 9×9=81 row-column constraint sets, 9×9=81 row-number constraint sets, 9×9=81 column-number constraint sets, and 9×9=81 box-number constraint sets: 81+81+81+81=324 constraint sets in all.

The end result of the reduction is a binary matrix, the exact cover instance of the Sudoku.

Since in a 9×9 Sudoku there are 81 cells and each cell has nine possible values ranging from 1 to 9, nine rows for each cell would be required of the matrix. Thus, we would be having a total of 729 rows in the matrix. At the same time, each constraint requires 81 columns each. Since

there are four constraints, there would be a total of 324 columns in the matrix. Therefore, we would be having a 729×324 binary matrix. To generate the Exact Cover matrix, an algorithm based on the four constraints will be formulated.

The matrix will then be evaluated using two methods: The naive implementation of Algorithm X and the implementation of Algorithm X with the use of Dancing Links technique. The results of these two reduction methods will also compared to each other after implementation.

Reduction of Sudoku with Clues to Exact Cover

Several methods are employed for reducing sudoku with clues. When a clue is provided, that clue is already chosen to be part of the solution. Other configuration of that clue are deemed useless. Some mechanisms to deal this include:

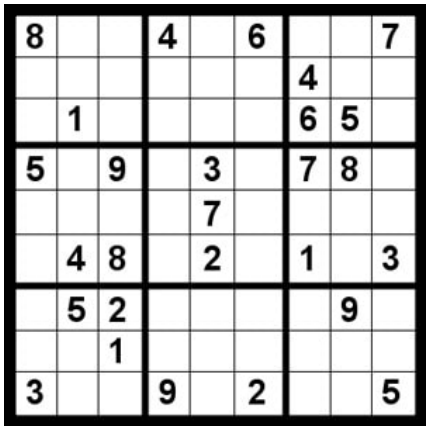
1. Exclusion of the excluded rows
2. Setting one or more of the four 1s into zero
3. Setting all values of the row into zero

The option one is the optimal choice since it will significantly reduce the size of the matrix, and improve the performance of the computation because these rows are not explored. However, need to still include the rows in order to have a uniform matrix size in all kinds of matrix in our samples. In this case, we use the option 3 above.

The position of 1s in each selected row from the exact cover matrix will define the position of solution in the Sudoku grid. For example, for $P_i = \{\ 5, 5, 7\ \}$ in the puzzle below, its representation in the matrix is shown as in the figure below.

$P_i = \{\ r, c, n\ \}$	**cell**	**row**	**column**	**box**
	1 ... 41 ... 81	**82 ... 124 ... 162**	**163 ... 205 ... 243**	**244 ... 286 ... 324**
$P_i = \{\ 5, 5, 7\ \}$	0 ... 1 ... 0	0 ... 1 ... 0	0 ... 1 ... 0	0 ... 1 ... 0

Table 4.1: Exact cover representation of $P = \{\ 5, 5, 7\ \}$



$\Rightarrow$

0001...0001000...00001000...001
0000...0000000...00000000...000
0000...0000000...00000000...000
0000...0000000...00000000...000
....
0100...0001000...00000010...001
0000...0000000...00000000...000
0000...0000000...00000000...000
0000...0000000...00000000...000

Figure 4.1: Sudoku Problem and its Exact Cover equivalent

Exact Cover Algorithms

After using the Brute-force search in solving Sudoku problems, the Sudoku problems will then be solved in its Exact Cover equivalent form. Exact Cover algorithms will be used in solving the problems. Knuth describes Algorithm X as the most obvious trial-and-error approach in (Knuth, 2000). It is a recursive backtracking algorithm that is generally useful in solving exact cover problems. Theoretically, the average search time complexity of Algorithm X is O(n). Another algorithm for solving exact cover problems is the Dancing Links. Dancing Links is the technique used by Donald Knuth to efficiently implement Algorithm X. The naive implementation of Algorithm X was observed by Knuth to take up an excessive amount of time in searching for 1s. To solve this problem, Knuth used a sparse matrix where only 1s are stored, thus, improving the complexity from O(n) to O(1).

DLX Implementation

After using Algorithm X to solve the Sudoku problem, this paper then proceeds to its most important implementation, using the Dancing Links(DLX) technique on Algorithm X in order to solve the Sudoku problem. The Dancing Links is a technique suggested by Donald Knuth to solve an exact cover problem using Algorithm X more efficiently. DLX concept is simply unlinking two connected nodes and also potentially link back again those two nodes. The idea of DLX is based on the observation that in a circular doubly linked list of nodes,

$L[R[x]] \leftarrow L[x];$

$R[L[x]] \leftarrow R[x];$

will remove node x from the list, while

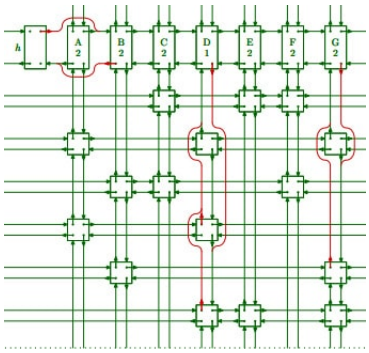
$L[R[x]] \leftarrow x;$

$R[L[x]] \leftarrow x;$

will restore xs position in the list, assuming that x.left and x.right have been left unmodified.

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	0	0	1	0	1	1	0
B	1	0	0	1	0	0	1
C	0	1	1	0	0	1	0
D	1	0	0	1	0	0	0
E	0	1	0	0	0	0	1
F	0	0	0	1	1	0	1

The above instance is implemented as linked list below. The DLX follows algorithm X as described in *Algorithm 2*. However, links are employed instead of a matrix. To delete column A (as in step 5.b of Algorithm X), the link from Header to A is re-assigned to B and the link from B to A is reassigned to Header as shown in the following figure. This is also the case to the nodes when removing rows (in step 5.a.i)



The links are restored using the second set of equations.

Interpretation of Exact Cover result

The position of 1s in each selected row from the exact cover matrix will define the position of solution in the Sudoku grid. For example, for X_j = { *6, 89, 215, 260* } as shown in its representation in the matrix, its equivalent Sudoku instance is P_i = { *1, 6, 8* } as shown in the Sudoku puzzle below.

cell	**row**	**column**	**box**
1 ... 6 ... 81	**82 ... 89 ... 162**	**163 ... 215 ... 243**	**244 ... 260 ... 324**
0 ... 1 ... 0	0 ... 1 ... 0	0 ... 1 ... 0	0 ... 1 ... 0

Table 4.2: X_j = { *6, 89, 215, 260* }

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 4.2: Completed Sudoku Puzzle

Research Schedule

The table below shows the list of activities and the corresponding time frame for this study.

Activities	**Date / Period**
Implementation of the design of the reduction process from Sudoku to Exact Cover	June 2017 to July 2017
Implementation of the Algorithm X and Dancing Links, Experimentation, and Research Report writing	July 2017 to Middle of August 2017
Final Research Presentation	Late August 2017
Revision and Finalization	Late August 2017 to Early September 2017
Bookbinding and Submission of Copies	Middle of September 2017

Table 4.3: Research Schedule from June 2017 to September 2017

References

Bertram Felgenhauer, F. J. (2006). *Mathematics of Sudoku I.* http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf.

Brian Bemman, D. M. (2015). *Exact Cover Problem in Milton Babbit's All-Partition Array.* http://www.titanmusic.com/papers/public/BemmanMeredithMCM2015.pdf.

Cazenave, T. (2012). *A search based Sudoku solver.* http://www.lamsade.dauphine.fr/~cazenave/papers/sudoku.pdf.

Dahlke, K. (2008). *Exact Cover.* http://www.mathreference.com/lan-cx-np,excov.html.

Gary McGuire, G. C., Bastian Tugemann. (2012). *There is no 16-clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem.* http://www.math.ie/McGuire_V1.pdf.

Kapanowski, A. (2010). *Python for Education: The Exact Cover Problem.* https://arxiv.org/pdf/1307.7042.pdf.

Knuth, D. (2000). *Dancing Links.* https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/0011047.pdf.

Mattias Harrysson, H. L. (2014). *Solving Sudoku efficiently with Dancing Links.* https://www.kth.se/social/files/58861771f276547fe1dbf8d1/HLaestanderMHarrysson_dkand14.pdf.

Mukherjee, S., Datta, S., Chanda, P. B., & Pathak, P. (2015). *Comparative Study of Different Algorithms to Solve N Queens Problem.* http://wireilla.com/papers/ijfcst/V5N2/5215ijfcst02.pdf.

Nivasch, G. (2004). *Solving Pentomino Puzzles with Backtracking.* http://www.cs.brandeis.edu/~storer/JimPuzzles/PACK/Pentominoes/LINKS/PentominoesNivasch.pdf.

Patrik Berggren, D. N. (2012). *A Study of Sudoku solving algorithms.* http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group6Alexander/report/PATRIK_BERGGREN_DAVID_NILSSON.rapport.pdf.

Sanjeev Arora, B. B. (2007). *Computational Complexity: A Modern Approach.* http://theory.cs.princeton.edu/complexity/book.pdf.

Sian Jones, P. R., Stephanie Perkins. (2007). *Properties of Sudoku Puzzles.*

Takayuki Yato, T. S. (2003). *Complexity and Completeness of Finding Another Solution and Its Application to Puzzles.* http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87

-2.pdf.

Wilson, R. (2006). *The Sudoku Epidemic.* http://www.ams.jhu.edu/~castello/150/handouts/SudokuEpidemic.pdf.