KTH Computer Science
and Communication

Solving Sudoku efficiently with Dancing Links

MATTIAS HARRYSSON
HJALMAR LAESTANDER

Degree Project in Computer Science, DD143X
Supervisor: Vahid Mosavat
Examiner: Örjan Ekeberg

Stockholm, Sweden 2014

Abstract

With this thesis, we hope to motivate software developers to seek out already existing solving algorithms instead of attempting to use a brute force algorithm or specialized solving algorithms.

The reason for choosing the Sudoku puzzle as a platform to demonstrate this is because it is well known around the world and easy to understand, while the reduction to an exact cover problem provides a challenge.

Because of the challenge in the reduction and because we did not find any earlier research which explained in detail how the reduction from a Sudoku puzzle to an exact cover problem is done, we decided to focus on that subject in this thesis. Using our previous knowledge in reduction and the information found during our research, the reduction was eventually solved.

Our conclusion is that Dancing Links is an effective solver for the exact cover problem and that a good reduction algorithm can greatly lower the solving time. The benchmarks also indicate that the number of clues in a Sudoku puzzle may not be the deciding factor of its difficulty rating.

Since the reduction to an exact cover problem was arguably the hardest part in this thesis, future research can hopefully make use of our detailed explanation of the reduction and use the time saved to explore other topics in more depth, such as the difficulty rating of Sudoku puzzle.

Sammanfattning

Att lösa Sudoku effektivt med Dancing Links

Med denna rapport så hoppas vi motivera mjukvaruutvecklare att söka efter redan existerande lösningsalgoritmer istället för att försöka använda en brute force-algoritm eller en lösningsalgoritm som är specialiserad på ett specifikt område.

Anledningen till att vi valde att använda Sudoku som ett verktyg för att demonstrera detta är för att det är känt runt om i världen och lätt att förstå, men också för att det är svårt att utföra en reducering till ett exakt mängdtäckningsproblem.

På grund av utmaningen i reduktionen och eftersom vi inte hittade någon tidigare forskning som detaljerat förklarade hur reduktionen från ett Sudokupussel till ett exakt mängdtäckningsproblem går till, bestämde vi oss för att fokusera kring det i denna rapport. Genom att använda vår tidigare kunskap inom reduktion och med den information vi hittade under informationssökningen kunde vi slutligen lösa reduktionen.

Vår slutsats är att Dancing Links är en effektiv lösare till det exakta mängdtäckningsproblemet och att en bra implementerad reduktion kraftigt kan sänka lösningstiden. Mätningarna visar också att antalet ledtrådar i ett Sudokupussel inte behöver vara den avgörande faktorn för sin svårighet.

Eftersom reduceringen till ett exakt mängdtäckningsproblem var den svåraste delen i vår rapport så hoppas vi att framtida forskning har användning av vår genomgång av reduceringen och istället kan använda den tiden till att utforska andra ämnen mer djupgående, som exempelvis svårighetsgraden för Sudokupussel.

Contents

Chapter 1

Introduction

Sudoku is a number puzzle originally created in the United States in 1979 and became popular in Japan during the 1980s. The puzzle was later introduced in 2004 by several British newspapers as a Japanese number puzzle. Shortly after the introduction, the puzzle became an international phenomenon. The name Sudoku derives from the Japanese word *Sūdoku* where *sū* means "digit" and *doku* means "single". [1] [2, p. 457]

The most common Sudoku puzzle is a *grid* of 81 cells divided into 9 rows and 9 columns. One *cell* can only contain a single integer between 1 and 9. The grid is also divided into 9 boxes where a *box* consist of 3 rows and 3 columns. Every new puzzle starts out with an arbitrary number of given integers placed in the grid. Every such integer is called a *clue*. The goal of the puzzle is to fill the remaining cells in the grid with integers such that every integer appears once in every row, column and box. Each cell that has yet to be filled has a number of possible integer values that are eligible for the cell, those are known as the *candidates* for that cell. [1] [3]

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

Figure 1.1: Sudoku Puzzle Challenge

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

Figure 1.2: Solution

For a grid to be considered valid, it is said that there can only exist one solution for a given grid with clues. This means that there can be only one way in which the integers in the grid can be placed to fill it, given the clues. The least known

number of given clues required for a unique solution is 17 [2, p. 457]. The solution in Figure 1.2 is an example of a unique solution to the challenge in Figure 1.1 which has 22 clues.

1.1 Problem Definition

The most basic way a computer can solve a Sudoku puzzle is to test all possible values for each remaining cell until a solution is found, or end without a solution when all values have been tested. This is referred to as a *brute force algorithm* or a *brute force search*. The name *brute force* is commonly used in many computer related subjects, but with the same fundamental idea — to test all possible values.

The problem with a brute force algorithm is performance. The brute force time complexity is exponential which is only feasible for small problem instances [4, p. 209].

This thesis will instead explore alternatives to a well implemented brute force algorithm. One approach would be to create a specialized algorithm specifically created with the purpose to solve a Sudoku puzzle. It could be argued that a better *first* approach would be to reduce Sudoku to another problem and use an already established and efficient solving algorithm instead. The implementation of that algorithm could also be reused for other problems while an implementation of a solver specifically made for the Sudoku problem cannot.

The process of reduction is explained in more detail in the background section. If the problem cannot be reduced to another problem efficiently or if the solving algorithm for the reduced problem does not perform good enough, then the *second* approach should be to create a specialized solving algorithm or a well implemented brute force algorithm.

The only previous work we could find that deals with reducing a Sudoku puzzle was "Python for education: the exact cover problem", the author Andrzej Kapanowski begins to explain how the reduction could be done to an exact cover problem. We found, however, the reduction to be incomplete because the article has more discussion on the solving algorithm of an exact cover problem rather than the reduction. There was no clear concept on how a solution to the Sudoku puzzle could be obtained once a solution to the exact cover problem was found.

Instead, the most important finding from this article was that the reduction from a Sudoku puzzle to an exact cover problem is possible but also that the exact cover problem could be efficiently solved by Algorithm X. This solving algorithm and the exact cover problem is explained in detail in the background section.

We have attempted to find other possible ways to reduce Sudoku puzzle as well as the existence of other solvers for the exact cover problem, but without success. Because of this we got motivated to focus our thesis on the reduction of a Sudoku puzzle to the exact cover problem in detail.

1.2 Problem Statement

- Can a Sudoku puzzle that is reduced to another problem be solved efficiently?
- Can different implementation of a reduction algorithm affect the solving time of a Sudoku puzzle?
- Is the number of clues the deciding factor in the solving time of a Sudoku puzzle?

1.3 Motivation

This thesis explores the reduction of a Sudoku puzzle to an exact cover problem and how an already existing exact cover solving algorithm finds a solution to it. The thesis explains the reduction algorithm in detail and will also seek out optimizations in the reduction algorithm to see if that can affect the solving time.

Consequently, the first goal is to put emphasis on the reduction rather than the solving algorithm for doing optimizations.

The second goal this thesis hopes to achieve is to motivate software developers to seek out already existing solving algorithms instead of "reinventing the wheel" with specialized solving algorithms for new problems. The idea is to fall back on creating new solving algorithms or using a well implemented brute force algorithm.

The reason for choosing the Sudoku puzzle as a platform is because it is well known around the world and easy to understand, yet the reduction to an exact cover problem is not as easy. The explanation of how the reduction is done in detail will in itself be useful for future works.

1.4 Structure

The thesis has been organized in the following way. The first section will introduce the reader to concepts necessary to understand the entirety of the thesis, as well as how the techniques and algorithm treated in this report function.

The second section gives a detailed description on how the reduction from a Sudoku puzzle to an exact cover problem that can be solved by Dancing Links can be done. This section then expands more on the same topic with regards to optimization and scalability.

The third section motivates which tools are used and the approach for conducting the tests for obtaining trustworthy results. The testing environment and what test data is used are also addressed.

After that, there will be a fourth section for presenting the obtained result. Interpretations on the presented results are discussed in the fifth section.

Following the discussion section, there will be a sixth section for concluding the problem statement based on the discussion.

Lastly, the references used in the thesis will be listed.

1.5 Terminology

Key terms used in this thesis.

Cell Individual square in a Sudoku puzzle, with or without a integer between 1 and 9.

Grid Another word for a Sudoku puzzle, both incomplete and complete. 9×9 block of cells.

Box 3×3 block of cells. Considered complete if containing 9 unique cells with integers between 1 and 9.

Clue Given integer between 1 and 9 for a cell in the grid. May not be modified in any way to complete the grid.

Candidates A cell which has yet to be filled by a integer. All possible integers which could be put in the cell are defined as candidates.

Chapter 2

Background

2.1 Complexity

2.1.1 Reduction

Reduction in this context is an algorithm used to transform one problem into another problem. Consider a "black box" which transforms problem X to problem Y, if we have an efficient algorithm to solve problem Y we can also solve problem X. However, without a black box the solving algorithm for problem Y is useless for solving problem X because it is on a form that the solving algorithm does not understand. The reduction is assumed to be efficient if it can be done in polynomial time. [4, p. 452]

2.1.2 NP-complete

NP (nondeterministic polynomial time) refers to the decision problems where the answer yes to the problems can be verified in polynomial time [5].

The hardest problems in NP are known as NP-complete. Learning that a problem is NP-complete is a good reason to stop looking for an efficient algorithm to solve it [4, p. 452]. The exact cover problem is an example of a NP-complete problem; it is one of Karp's 21 NP-complete problems.

Sudoku is NP-complete when generalized to a n×n grid according to Mária Ercsey-Ravasz and Zoltán Toroczkai in their report "The Chaos Within Sudoku". [6, p. 1] It is important to understand that a standard 9×9 Sudoku is *not* NP-complete because it is a finite instance, the classification of NP-complete is only for a generalized n×n Sudoku.

2.2 Exact Cover Problem

The exact cover problem is a decision problem to find an exact cover. Given a set S and another set where each element is a subset to S, is it possible to select a set of

subsets such that every element in S exist in *exactly one* of the selected sets? This selection of sets is said to be a *cover* of the set S. [7, p. 4]

This problem can be visualized as a binary matrix M. In M, does it exist a set of rows such that every column in the rows contain exactly one 1? [8, p. 2] [9, p. 2]

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array} \tag{2.1}$$

This matrix can be defined to have a universe $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ where every row is a subset of S. Each row has a 1 in the column which describes the column number in S. For instance $A = \{1, 2, 5, 7\}$ because the row has a 1 in column 1, 2, 5 and 7.

After a quick overview of M in (2.1), we can see that $B \cup C \cup E$ creates a set where a 1 appears exactly once in every column. For our universe S this would be $B = \{3, 4, 6\}$, $C = \{1, 5, 9\}$ and $E = \{2, 7, 8\}$ where every element in S appears exactly once in B, C and E. The selected three rows are said to *cover* S.

This binary matrix representation of the exact cover problem is important to understand because this representation is what Donald Knuth uses for his Algorithm X. In a later section, the Sudoku will need to be reduced to a binary matrix for it to be compatible with the solving algorithm used in this thesis.

2.3 Algorithm X

Algorithm X is a nondeterministic, recursive algorithm that uses depth-first search for backtracking [9, p. 2]. It was created by Donald Knuth and can be used to find all solutions to the exact cover problem [8, p. 3].

The algorithm is based on a very simple technique which Knuth think should be better known. Given a node x which points to two elements in a doubly linked list, $L[x]$ points to the left element of x and $R[x]$ points to the right element of x. For instance, $L[R[x]]$ is pointing to x if the right element of x is pointing its left element back to x. This is more easy to see if we let $y = R[x]$ such that $L[R[x]] = L[y]$, which describes the left element of y.

The following describes two operations on x, first (2.2) removes x from the doubly linked list and (2.3) inserts x back into the doubly linked list.

$$L[R[x]] \leftarrow L[x], \quad R[L[x]] \leftarrow R[x] \tag{2.2}$$

$$L[R[x]] \leftarrow x, \quad R[L[x]] \leftarrow x \tag{2.3}$$

The first operation should be very obvious but the second operation is what Knuth think is not well known. The insertion works because x still has the references it had before it was removed from the doubly linked list.

It would seem pointless to insert an element that was just removed but Knuth argues that there are several applications where this is useful; the most notable usage is for backtracking.

2.4 Dancing Links

Dancing Links, or DLX for short, is the technique suggested by Donald Knuth for implementing Algorithm X efficiently. Given a binary matrix, DLX will represent the 1s as *data objects*. Each data object x have the fields $L[x]$, $R[x]$, $U[x]$, $D[x]$ and $C[x]$. The fields are for linking to any other cell with an occupying 1 to the left, right, up and down. Any link that has no corresponding 1 in a suitable cell will link to itself instead. [8, p. 5]

The last field is a link to the *column object* y which is a special data object that has two additional fields, $S[y]$ and $N[y]$, which represents the column size and a symbolic name chosen arbitrarily. The column size is the number of data objects that are currently linked together from the column object. Each row and each column is a circular doubly linked list, and if a data object is removed from the column, the size is decremented. Remember that the removed data object still has the links pointing as they did before the data object was removed. [8, p. 5]

$$\begin{matrix} A & B & C & D & E & F & G \end{matrix}$$
$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

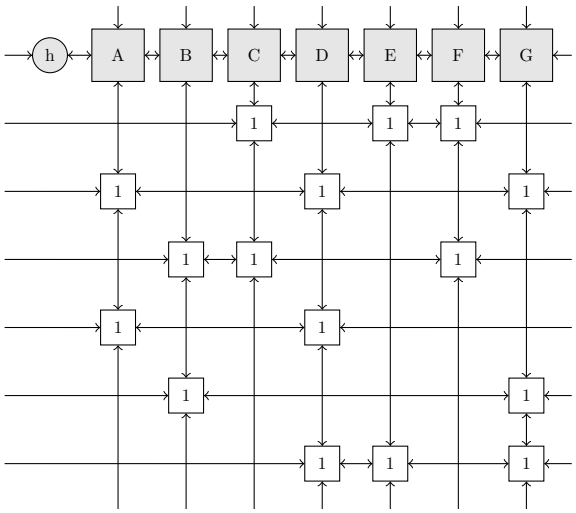


Figure 2.1: Binary matrix representation in DLX

Figure 2.1 visualize how a given binary matrix is represented in the DLX data structure. It shows how each 1 in the matrix is represented by a data object with links to any nearby data object. The arrow head at the data or column objects

describes a link; also note how the links have a circular connection for both the row and columns.

Each column has a symbolic name from A to G where h is the root column object. The root column object is a special case column object which uses the same data structure but cannot link to any data objects directly; it is only used by DLX as an entry point to the data structure.

We define a function `search(h, k, s)` where `h` is the root column object, `k` is the current depth and `s` is the solution with a list of data objects. The function should be invoked with `k = 0` and `s = []`. If `s` is a linked list the `k` can be omitted.

```
1  search(h, k, s) =
2      if R[h] = h then
3          print_solution(s)
4          return
5      else
6          c ← choose_column_object(h)
7          r ← D[c]
8          while r ≠ c
9              s ← s + [r]
10             j ← R[r]
11             while j ≠ r
12                 cover(C[j])
13                 j ← R[j]
14             search(h, k + 1, s)
15             // Pop data object
16             r ← s_k
17             c ← C[r]
18             j ← L[r]
19             while j ≠ r
20                 uncover(C[j])
21                 j ← L[j]
22             r ← D[r]
23         uncover(c)
24         return
```

Listing 2.1: DLX

How the solution is printed will be processed in the upcoming section for reducing Sudoku, which is specific for a Sudoku solution. Add a function pointer argument to `search` in order to have more options on how the solution is printed

On row 6 in Listing 2.1, the column object is chosen which can be implemented in two different ways, either by choosing the first column object after the root column object, or by choosing the column object with the fewest number of 1s occurring in a column. Choosing the latter is argued by Donald Knuth to minimize the branching factor. [8, p. 6]

The other two functions `cover` and `uncover` will for a specified column object c either cover c or uncover c. Listing 2.2 shows the cover operation that removes c from the doubly linked list of column objects and removes all data objects under c.

The function will make use of the operation (2.2) introduced along with Algorithm X.

```
1  cover(c) =
2      L[R[c]] ← L[c]
3      R[L[c]] ← R[c]
4      i ← D[c]
5      while i ≠ c
6          j ← R[i]
7          while j ≠ i
8              U[D[j]] ← U[j]
9              D[U[j]] ← D[j]
10             S[C[j]] ← S[C[j]] − 1
11             j ← R[j]
12         i ← D[i]
```

Listing 2.2: Cover function

The other function `uncover` in Listing 2.3 is more interesting because it make use of the (2.3) list operation, which Knuth wanted to be better known. [8, p. 1]

```
1  uncover(c) =
2      i ← U[c]
3      while i ≠ c
4          j ← L[i]
5          while j ≠ i
6              S[C[j]] ← S[C[j]] − 1
7              U[D[j]] ← j
8              D[U[j]] ← j
9              j ← L[j]
10         i ← U[i]
11     L[R[c]] ← c
12     R[L[c]] ← c
```

Listing 2.3: Uncover function

The insertion operation in (2.2) is undone by doing the reverse order of the covering operation in `cover` because (2.3) "undo" the operation (2.2) in Listing 2.2. The rows were removed from top to bottom and instead must be added from bottom to top to undo the operation. The same principle applies for the columns that were removed from left to right and instead must be added from right to left to undo the operation.

2.4.1 Solving Steps

The matrix in Figure 2.1 represents an exact cover problem and we assume that we have a root column object h that describes this matrix; we will now explore how DLX will find a solution to this problem. This example is based on Knuth's

example with the hopes to clarify some misconceptions [8, p. 7].

DLX is invoked with `search(h,0,[])`, and since $h \neq R[h]$, column A is chosen as the next column object to be covered. Figure 2.2 shows how the first two rows in A is removed. This will affect the data objects in D and G because these two column objects also have data objects on this row. The dotted line in Figure 2.2 shows how the new links are formed; note how the arrows are still pointing from the data objects that are on the removed rows.

Following DLX in Listing 2.1, we know that r points to the first data object in A, so the next columns to be covered are D and G, as seen in the loop on row 11. The result of this can be seen in Figure 2.3, where the thick lines displays the new links.

Next, the branch uses `search(h, 1, s)` to search after all column objects that were covered in the first row in A. This will cover column B and leave no 1s in column E, which in turn will leave `search(h, 2, s)` without a solution. This will return the algorithm to Figure 2.3, and the algorithm proceeds to the first row in column object A.

$$\begin{matrix} B & C & D & E & F & G \end{matrix}$$
$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

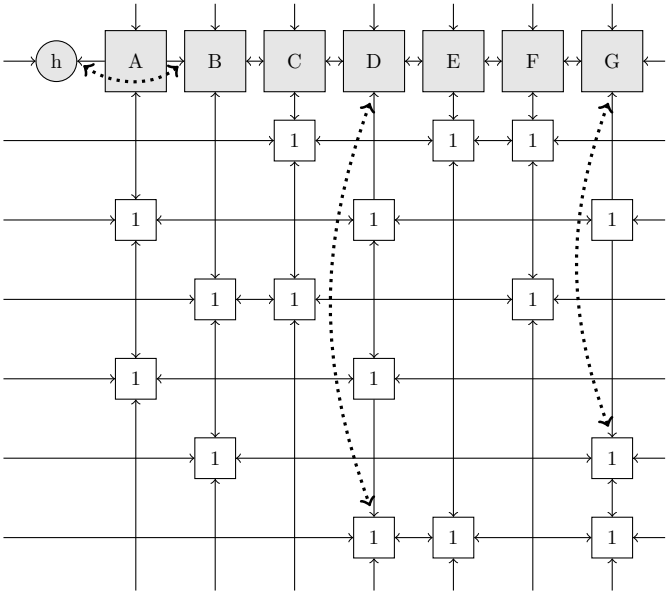


Figure 2.2: Column A in Figure 2.1 has been covered

$$\begin{array}{cccc} B & C & E & F \\ \left(\begin{array}{cccc} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{array}\right) \end{array}$$

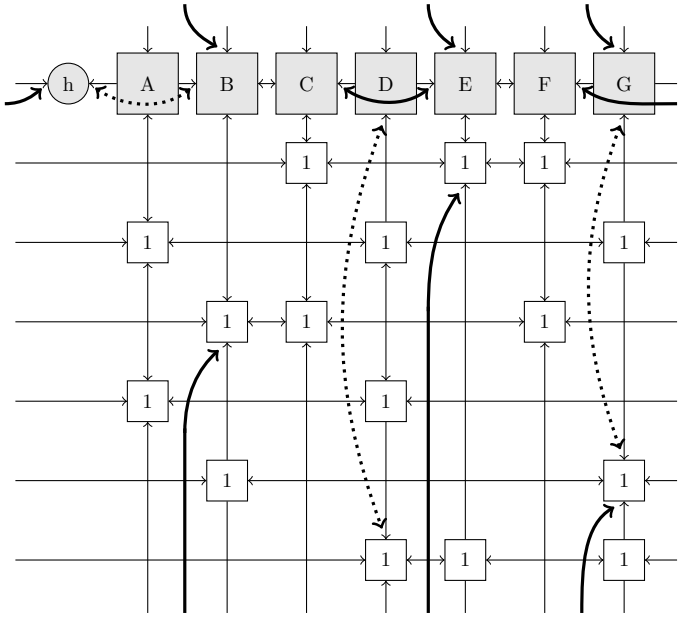


Figure 2.3: Column D and G in Figure 2.2 has been covered (row 1 and 3)

Eventually the solution is found, which can be various in different ways. Given the solution s, Knuth prints the rows containing $s_0, s_1, \ldots, s_{k-1}$ where each s_α for $\alpha \in \{0, 1, \ldots, k-1\}$ is printed using the algorithm in Listing 2.4.

```
1  print(N[C[s_α]])
2  i ← R[s_α]
3  while i ≠ s_α
4        print(N[C[i]])
5        i ← R[i]
```

Listing 2.4: Printing the solution

The solution for this problem can be printed as

```
A D
B G
C E F
```

where `A D` is for row 4, `B G` for row 5 and `C E F` for row 1. This solution is unique and it is easy to verify that row 1, 2 and 5 provides a cover for the exact cover problem presented in Figure 2.1.

2.5 Difficulty Rating

A variety of definitions of the difficulty ratings was found during our search for information. Some reports have assumed that fewer clues lead to harder puzzles, which is understandable considering that the matter of difficulty ratings have not been widely researched in the scientific department. [10, p. 2]

The ratings are however discussed in different Sudoku forums and websites with the conclusion that the difficulty of Sudoku in fact does not depend on the number of clues, but instead it is argued that the difficulty is determined by the number of different techniques and decisions needed, and the difficulty of these techniques. The techniques in question are the different techniques that humans use for solving Sudoku, which have different levels of difficulty. [11] [12]

Chapter 3

Reducing Sudoku

One of the most important parts in this thesis is the algorithm for reducing a grid to an exact cover problem. Without understanding this reduction, the Dancing Links algorithm cannot be used for the purpose of solving a grid.

The rules of Sudoku were described in the introduction and every rule can be described as a constraint.

Cell Each cell can only contain one integer between 1 and 9.

Row Each row can only contain nine unique integers in the range of 1 to 9.

Column Each column can only contain nine unique integers in the range of 1 to 9.

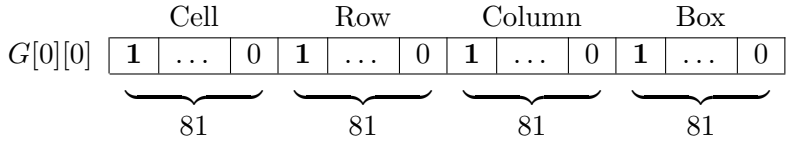
Box Each box can only contain nine unique integers in the range of 1 to 9.

Consider a grid G with at least 17 clues and with a unique solution. The clues will have an effect on all four constraints since a clue cannot be moved or changed. The relationship between the clues will then decide in which cells in the row, column and box the remaining integers can be placed. In other words, the clue determines the candidates for the remaining cells on the same row, column and box.

The reduction from the grid G must preserve the constraints in a binary matrix M. In M there must then exist a selection of rows such that a union between them covers all columns, otherwise there is no solution.

This reduction does not seem plausible at first glance since a binary matrix can only contain 1s and 0s, and a grid can only contain integers between 1 and 9. It also does not seem plausible to preserve the constraints for all cells.

What makes this reduction possible is having each row in M describing all four constraint for each cell in G. When a row is chosen from M as part of the solution, what is actually chosen is an integer which complies with all four constraint for the cell in G that the row describes. Each cell in G creates rows as follows



though the constraints can come in any order as long as they are consistent for all rows. The following sections describes how each constraint is preserved in M.

3.1 Cell Constraint

Each cell in G has nine candidates which means there must be nine rows in M for each cell. One of these nine rows must be included for the solution for each cell. Since the solution must cover all columns, each of the nine rows has 1s in their own column. The first cell has 1s in the first column for the first nine rows, the second cell has 1s in the second column for the next nine rows and so on for the remaining cells. This will force the algorithm to always include at least one of the rows for each cell to cover all columns. Since there are 81 cells, there are 81 columns required for the cell constraint, and since each cell requires nine rows, M must have space for its $9 \cdot 81 = 729$ rows.

	1	2		81
$G[0][0]$ 1	**1**	0	...	0
⋮				
$G[0][1]$ 1	0	**1**	...	0
⋮				
$G[8][8]$ 9	0	0	...	**1**

3.2 Row Constraint

Each row in G can only have a set of integers between 1 and 9. To preserve this constraint in M, the 1s are placed in a different pattern than in the cell constraint. For 9 cells to comply with a row constraint, we need to place the 1s for *one* row in G over 9 rows in M.

For one cell in G the 1s are placed in a new column for each of the nine rows in M. This is repeated in the *same* columns for the first nine cells in G or the 81 first rows in M. The next row constraint starts in the 10th cell or on row 82, but the 1s are placed starting after the last column used by the first row in M. This will force the algorithm to only pick a unique integer value for each cell in the row since the columns with 1s spans the same columns for the cells on the same row, but only one column can be in the solution.

	1	2		80	81
$G[0][0]$ 1	**1**	0	...	0	0
$G[0][0]$ 2	0	**1**	...	0	0
⋮					
$G[0][1]$ 1	**1**	0	...	0	0
$G[0][1]$ 2	0	**1**	...	0	0
⋮					
$G[8][8]$ 8	0	0	...	**1**	0
$G[8][8]$ 9	0	0	...	0	**1**

3.3 Column Constraint

Like the rows in G, the columns in G can only have a set of integers between 1 and 9. To preserve this constraint from G in M, the 1s are again placed in a different pattern in M.

For every cell in G the 1s are placed in a new column for each of the nine rows in M. This is repeated for the second cell, though instead of reusing the same columns like for the row constraint, the 1s are starting where the previous cell ended. This occurs for the first nine cells. For the upcoming nine cells, the pattern starts at the first column again. This will pair up the columns of 1s in M with each cell in a column in G. This is going to force the algorithm to only pick a unique integer value for each cell in the column of G.

	1	2	3	4	5	6	7	8	9	10	11		81
$G[0][0]$ 1	**1**	0	0	0	0	0	0	0	0	0	0	...	0
$G[0][0]$ 2	0	**1**	0	0	0	0	0	0	0	0	0	...	0
⋮													
$G[0][1]$ 1	0	0	0	0	0	0	0	0	0	**1**	0	...	0
$G[0][1]$ 2	0	0	0	0	0	0	0	0	0	0	**1**	...	0
⋮													
$G[1][0]$ 1	**1**	0	0	0	0	0	0	0	0	0	0	...	0
$G[1][0]$ 2	0	**1**	0	0	0	0	0	0	0	0	0	...	0
⋮													

3.4 Box Constraint

The box constraint will follow, just like the previous three constraints, a certain pattern. Each box contains three rows and three columns; this will create what would seem to be an irregular pattern at first.

The first three cells in G will share the same columns of 1s in M. The next three cells in G will share the same columns of 1s, and the last upcoming three cells for that row will also share the same columns of 1s. This is because the nine cells on

the first row in G are located in three different boxes. The same pattern is repeated for all rows in G.

This will create the constraint for three complete boxes. Their columns are equal in M if they are in the same box in G. For one cell in G the columns are placed just like in the row constraint in M, first a 1 in the first column, then a 1 in the second column for the second row in M and so on until the next cell in G. This happens because the integers in the box must have nine unique integers ranging from 1 to 9. This will force the algorithm to once again select rows based on the constraint in the grid.

This was the last of the constraint. Each constraint will require 81 columns each, which leads to that the total number of columns required in M is $81 \cdot 4 = 324$.

	1	2		10	11		19	20		81
$G[0][0]$ 1	**1**	0	...	0	0	...	0	0	...	0
$G[0][0]$ 2	0	**1**	...	0	0	...	0	0	...	0
⋮										
$G[0][1]$ 1	0	0	...	**1**	0	...	0	0	...	0
$G[0][1]$ 2	0	0	...	0	**1**	...	0	0	...	0
⋮										
$G[0][2]$ 1	0	0	...	0	0	...	**1**	0	...	0
$G[0][2]$ 2	0	0	...	0	0	...	0	**1**	...	0
⋮										
$G[1][0]$ 1	**1**	0	...	0	0	...	0	0	...	0
$G[1][0]$ 2	0	**1**	...	0	0	...	0	0	...	0
⋮										

3.5 Obtaining a Solution

Obtaining the solution to the grid is quite easy but, requires a slight modification of DLX. Each data object must also include a field for storing the row placement in the original binary matrix M, because when M is transformed by DLX into the links, there is no way of knowing from the found solution which data object is for which row in M.

When a solution is found, the list of data objects must be sorted in descending order based on their row number. The solution should have 81 rows, one for each cell in the grid that is being solved. Since our domain is 1 to 9, we will use modulus 9 with every row. Before applying modulus, we must increment the row number with one since the implementation deals with zero-based indices while the solution does not.

Thus the solution found by DLX can because of this be easily transformed back into the original grid again but without any unknown cells. The grid can then be verified if it complies with the constraints or not.

3.6 Optimization

It is known that M must be of size 729×324 to store all possible constraints. Each given clue only requires one row in M because we want to force the algorithm to only pick these rows for the solution. For unknown cells we must add rows for each candidate. All rows in M are not required to present the constraint, but if rows are removed from M the structure of M is lost and the solution cannot be obtained. Because of this, the rows that are not needed are instead represented as a row of 0s. This will not affect the solution in any way while still keeping the row numbers in M.

The problem with this approach is that we are only interested on how the 1s are placed, but the majority of elements in M are 0s. There are a lot of unnecessary iterations when creating the links because the 0s are skipped.

We propose a better alternative for creating the links. Instead of creating the binary matrix M, the links are created directly. Since the number of columns is always the same and since all columns are needed, the 324 column objects are created and stored in an array. The column objects are still linked together, but the access time to all column objects are now O(1).

For each new cell in G we create four data objects and link them together. Every new cell would be a row further down in M, so the four created data objects can instead be appended to the column objects. All insertion operation takes O(1) because it is just a matter of changing the pointers, and all column objects can be accessed in O(1) through the array. This means that every new cell takes O(1) to insert as links.

Note that this is not a reduction to an exact cover problem but rather a reduction directly to DLX.

3.7 Scalability

The explained reduction is for a standard 9×9 Sudoku, but the same principle can be used for smaller or larger grids. The only change is the number of rows and columns since the constraints are still the same. For example, hexadecimal Sudoku with a 16×16 grid will give us a binary matrix of size 4096×1024. Since the binary matrix has almost 18 times more elements, the optimization could be even more important, but the purposed optimization should have no problem of being applied for any grid of any size.

Chapter 4

Method

The programming language of choice is *Python* which is a very high level object-oriented, interpreted and interactive programming language [13]. Python was chosen from experience since it transforms ideas into code fast but also for its great support in mathematics, science and engineering offered with SciPy. SciPy is an open source library of scientific tools such as *numpy* which is widely used in the implementation of algorithm since it brings good support for working with matrices (n-dimensional arrays). SciPy also offers support for plotting 2D graphs using Python syntax with the *Matplotlib* package and all 2D graphs in the upcoming result section is generated using Matplotlib. [14]

The downside of using Python compared to a compiled language such as C is the same downside as with all interpreted languages which is performance. However, the performance downside needs to be put in perspective. Solving a grid is not as performance critical as for example rendering frames for a game where slow renderings can ruin the game experience to the point of making the game unplayable. Using a Sudoku solving algorithm to find the unique solution for a given grid is more acceptable to more delays. However, this thesis will solve thousands of grids several times for benchmarks which can quickly scale in execution time to the point where Python can become an infeasible alternative.

The primary reason for still choosing Python despite its downside is because performance heavy functions can be implemented using C, this means that if there is a performance issue the functions can be profiled and replaced accordingly. Having this scalability option with SciPy is why Python is the language of choice.

4.1 Test Data

4.1.1 Storage and Input

Since Python is an interpreted language the option to store test data as data structures in Python files is available but little is gained from this approach. The biggest downside is that test data becomes tightly coupled with Python which limits the room for extending the study for future works or usage in other works. Another

downside is that existing test data provided by others are already in a accessible form since a complete or incomplete grid only needs a handful of integers as input. The most common way observed is to store the grid as a line in a text file where each character represents a cell for the grid. Cells without clues are represented with a dot to maintain the structure of the grid.

For example, the Sudoku Puzzle Challenge in Figure 1.1 would be stored as one line like this

```
........82.1.........9..673....5....9.7..3.41....7.....5....289.....5...36..4....
```

where every ninth character is the last cell for a row in the grid it represents.

The text files can also store multiple grids where every line in the text file represents a grid. The text files can be categorized based on for instance difficulty level, structure, number of clues and so on.

The program can with this expected input easily transform the line of characters into a data structure that the algorithm understands and doing so automatically for each new line in the text file.

4.1.2 Availability

Several thousands of grids stored in text files on the form as describes in previous section can be found using any online search engine. They can all be used as test data as long as they provide a unique solution.

The observed grids have a mixture of different number of given clues but also sometimes the grids comes with a difficulty rating label such as *easy*, *medium* or *hard*.

The most notable finding is all possible grids with 17 clues which amount to 49151 grids. For grids with 18 or more clues a total of 14236 grids was found but the majority of those grids have between 21 and 31 clues. For the grids with 18 or more clues the majority was also claimed to be of hard difficulty rating with a few claimed to be easy. Note that this difficulty rating is in regards for human solvers. See Figure 4.1 for the distribution for the grids with 18 or more clues.

4.1.3 Selection

We have chosen to use the found grids with 17 to 31 clues for doing benchmarks and analyzing the alleged difficulty rating for our solver in order to be as comprehensive as possible. We think that this mix of different grids provides good conditions in order to answer the problem at hand.

The reason for discarding the grids with 32 to 81 clues is because of the lacking number of found grids. The exception is made for the grids with 19 and 20 clues for having a coherent rise of clues in the interval. We will be careful when analyzing the results in the regards of the grids with 19 and 20 clues.

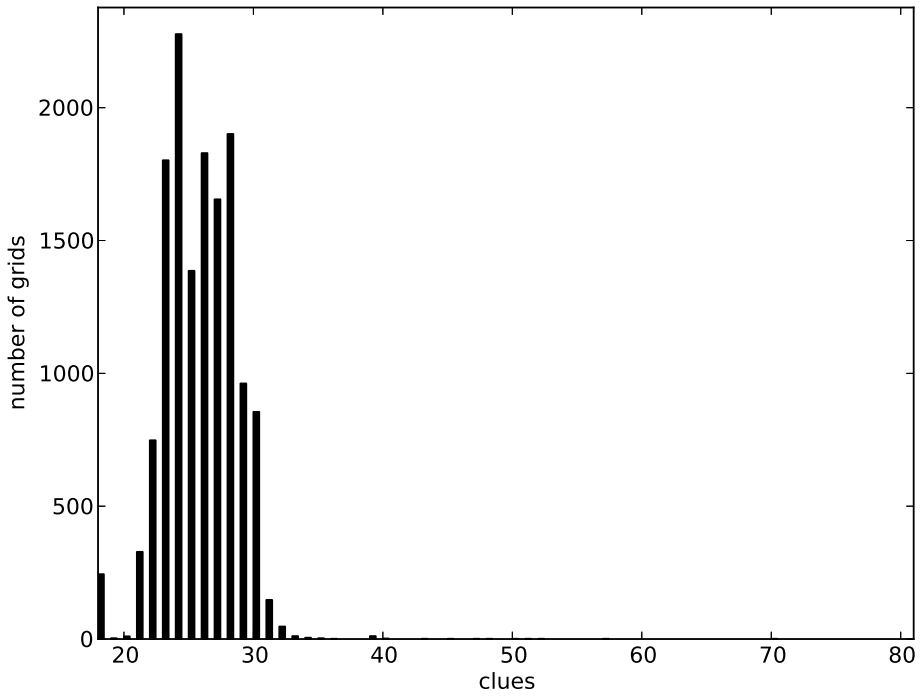


Figure 4.1: Distribution of grids with 18 to 81 clues

4.2 Testing Environment

All tests are performed using the same hardware and software setup for consistent results with only essential programs running.

4.2.1 Hardware

Motherboard Gigabyte GA-P55-UD4

CPU Intel Core i5 @ 2.65 GHz (first generation, quad core)

GPU AMD Radeon HD 5870 (open source drivers)

Memory 2 × Corsair Dominator DDR3 2048 MB @ 1333 MHz

Storage Intel 510 Series SSD 120 GB (ext4 file system)

4.2.2 Software

Operating system of choice is 64-bit GNU/Linux running on Linux kernel 3.11.0-18-generic with the XFCE 4.10 desktop environment. All tests are executed from

the XFCE terminal emulator. The motivation for using this operating system is because it is free as in both price and freedom so that anyone can reproduce all results in this thesis.

4.3 Measuring Time

Measuring time is tricky but Python comes with "battery included" and the *timeit* module is an example of that; it is a module for measuring time reliably. The pitfalls when measuring time are subtle but can still affect the results considerably. For example, to keep Python simple and easy to use the authors have decided to include a garbage collector (GC) which handles memory de-allocation for the programmer, but this de-allocation process can start running at any point during program execution and interfere with the timing. Using timeit avoids this pitfall since it has the GC disabled by default during time measurements. [15]

The default timer which timeit uses for doing measurements has different precision depending on platform. Since GNU/Linux is used, the UNIX "wall clock time" is utilized as the default timer, which means that any background process running will interfere with the timing. The documentation still recommends using this timer for benchmarking algorithms [16]. [15].

The last point for measuring time reliably is doing the same test more than once. In this context which deals with solving given grids, the time measured would be the solving time, which would be done n number of times. The total running time for solving the grid n number of times is noted. This process is then repeated m number of times. The noted times can be stored in an array of m elements and the elements will not be equal but vary. The pitfall here is to take the average time from all these elements and think that is the "best result". The running time vary because of background processes or other interfering factors, and not because the code takes longer to execute. If the times vary too much, the test should be discarded. Otherwise, the *minimum time* should be considered as the "best result" and the rest discarded, unless such observations provide interesting results.

Chapter 5

Result

5.1 Reduction

The naive reduction can both be reduced to an exact cover problem and then further to a root node used by DLX to find a solution. However, the optimized reduction can only be reduced to a root node, which is why these sections' benchmarks are measured from the start of the reduction of a given grid of clues to a root node.

Figure 5.1 shows grids from 81 clues (complete) to 17 clues, which is the least known number of clues for a unique solution. This will give us the opportunity to observe how the two reductions scale regarding the number of given clues.

Each grid is reduced 10 times where the total running time (in seconds) is measured. This procedure is repeated three times and represented with lines r1, r2 and r3 for the naive and the optimized reduction in Figure 5.1.

Table 5.1 gives a good overview of the total runtime for each repeat but also a approximate time on how long each test took to perform.

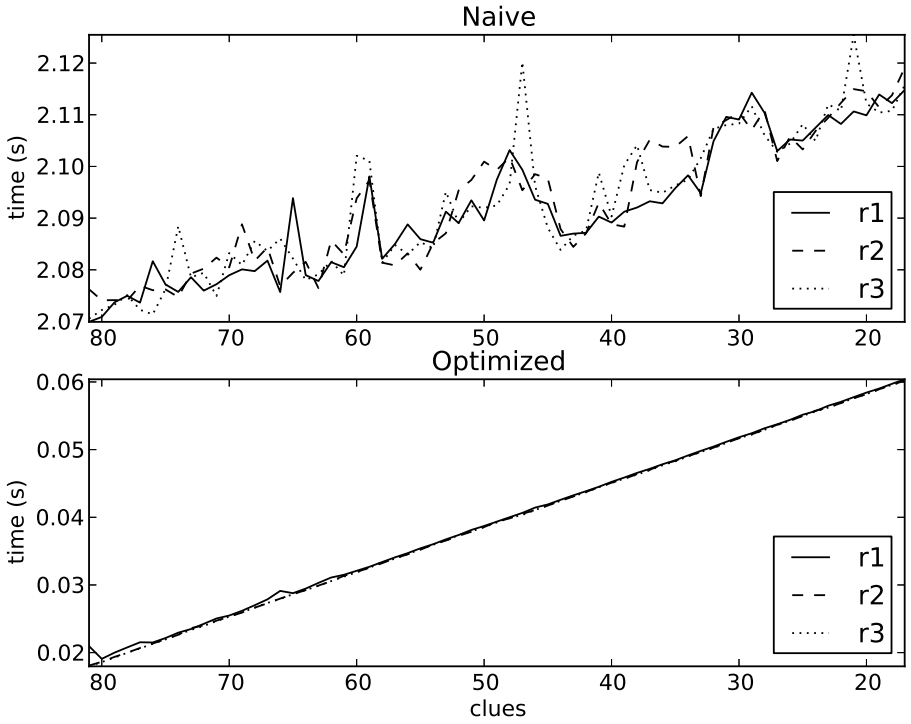


Figure 5.1: Reducing grids with descending number of given clues to DLX

Table 5.1: Runtime for each repeat in seconds

Naive	
r1	135.944489956 s
r2	136.038166285 s
r3	136.031737089 s
Total	≈ 408 s

Optimized	
r1	2.56227779388 s
r2	2.54506421089 s
r3	2.54371905327 s
Total	≈ 7.65 s

5.2 Difficulty

The difficulty rating of grids have been discussed and mentioned throughout this thesis. For exploration on this topic, benchmarks for our selection of grids have been conducted and presented in Figure 5.2.

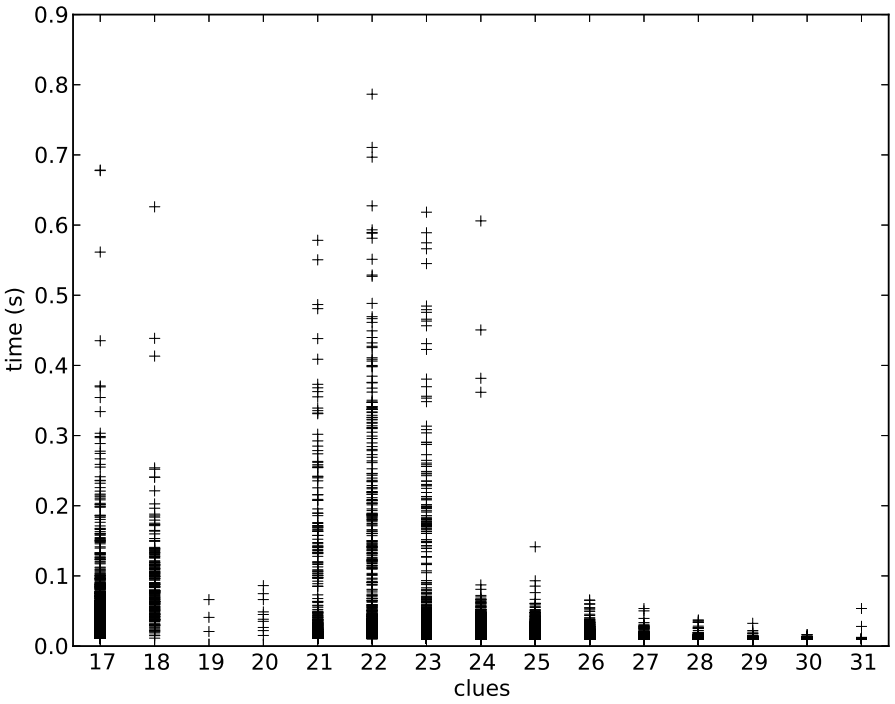


Figure 5.2: Solving time for grids with 17 to 31 clues

Each grid in Figure 5.2 have been measured from the start of the reduction to when a solution is found. The validation time was not measured. This was done before conducting the benchmark to ensure that each grid was valid and had a unique solution.

The plotted time is the best running time for each grid after doing the process 10 times with 3 repeats.

Figure 5.3 will give a more detailed overview on the solving time distribution opposed to Figure 5.2. The x-axis represents all grids with 17 clues in no particular order.

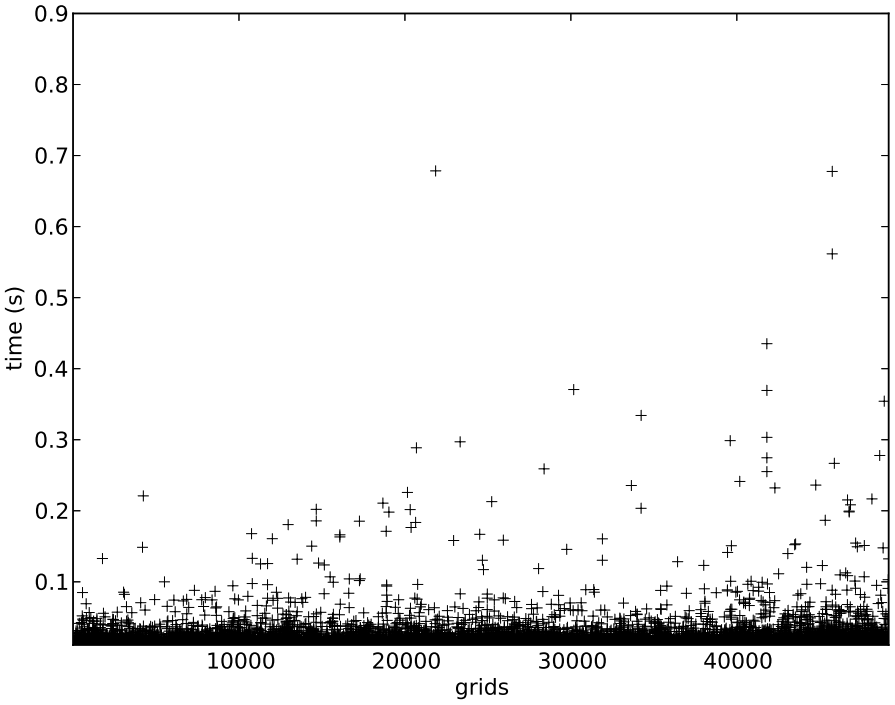


Figure 5.3: Solving time distribution for all grids with 17 clues

Chapter 6

Discussion

6.1 Reduction

The result for the reduction in Figure 5.1 show that the optimized implementation is considerably faster than the naive implementation. The benchmark for one specific grid gives an improvement of over two seconds. An implication of this is the possibility that different placement of the clues will affect the result as is the case if a different grid is used. A further study with more focus on using additional grids is therefor suggested.

Another interesting finding was how uneven the naive implementation in each repeat was with regards to the running time, as opposed to the optimized implementation. A possible explanation for this result is that the extended time that the naive implementation runs on gives more room to be affected by background processes. It can thus be suggested that using the optimized implementation would give the best results when benchmarking different grids in solving time because of the even reduction benchmarks.

Surprisingly, the time differences between reducing 81 clues and 17 clues were almost identical between the two implementations, and lower than expected. This could mean that benchmarks for grids with different number of clues and difficulty labels can be determined by the elapsed time of the solving algorithm and not by the reduction.

The reduction time for all clues in the naive implementation took longer to solve than the majority of Sudoku puzzle with the optimized implementation and DLX as seen in Figure 5.2. This further strengthens the importance of optimizing the reduction rather than the solving algorithm.

6.2 Difficulty

An initial objective of this report was to identify if the number of clues determines the difficulty of a grid. It is apparent from Figure 5.2 that the grids that have the longest solving time were not grids with 17 clues, but grids with 22 clues.

The majority of grids were solved below 0.1 seconds, but the spread of the plots are consistently thicker for grids with 22 clues than for any other grid above 0.1 seconds. This suggest that 22 clues have an optimal number of clues because the number of clues affect the reduction time as seen in Figure 5.1, in turn this also affect the overall solving time. This also seems to give enough clues to be placed in relation to each other such that the DLX has difficulties to find a solution.

The affect from the placement of clues makes sense if we think about it for a moment. It is presumably impossible for a grid with say 50 clues or more to be more difficult than a grid with 17 clues because there are only so many ways that the clues in a grid can be placed. So even though the number of clues may not be the only determining factor for the difficulty among grids, it nevertheless affects it to a certain degree.

It should be noted that there were significantly more test data for 17 clues than for 22 clues, 49151 grids compared to 758, but the grids that took the longest to solve still belong to the collection with 22 clues. One possible explanation for this is that the majority of grids with 18 clues or more were labeled with hard difficulty rating. This could be explored in further studies with the same number of grids, but instead use grids with a label of easy difficulty rating. If any correlation between the difficulty ratings and running time is found, then it could be possible to come up with a more accurate explanation.

Figure 5.3 show the distribution of the solving time for all grids with 17 clues in greater detail scattered in the region of 0 to 0.7 seconds. This further reinforce that the number of clues have less affect on the solving time. If the clues were the only factor that affected the solving time, the spread would be evenly distributed in a horizontal line.

Together, the findings from Figure 5.2 and Figure 5.3 supports the hypothesis found on different Sudoku websites, that the difficulty does not reside in the number of clues, but on something else.

Chapter 7

Conclusion

Returning to the problem statement at the beginning of this thesis, it is now possible to state that DLX solves Sudoku puzzle efficiently. What is efficient is highly subjective, but using an interpreted programming language and solving all gathered test data within one second and solving the majority of that test data under 0.1 seconds is what motivates our conclusion.

We can also conclude that our optimized implementation of the reduction has greatly reduced the solving time of a Sudoku puzzle. The downside of using the optimized implementation is that no other solving algorithm to the exact cover problem but DLX can be used after the reduction.

The results suggest that the solving time for using our own solver is not determined by the number of clues as discussed in the previous section. What we cannot conclude is that the difficulty is determined by the number of different techniques and decisions needed, as stated on various Sudoku websites, since DLX is not based on human solving techniques.

Our first goal, to put emphasis on the reduction rather than the solving algorithm for doing optimizations, proved to be a viable decision concidering the result we obtained.

Regarding the second goal, to motivate software developers to seek out already existing solving algorithm. We have demonstrated that it is possible to use an already existing solver to solve the problem efficiently, and by that we hope to have achieved our second goal but we cannot conclude if it has been reached.

Bibliography

[1] Nationalencyklopedin. Sudoku [homepage on the internet]. [cited 2014 February 10]. Available from `http://www.ne.se/sudoku`.

[2] Shanchen Pang; Eryan Li; Tao Song; Peng Zhang. Rating and generating sudoku puzzles. *Second International Workshop on Education Technology and Computer Science*, 3:457–460, 2010.

[3] Gaby Vanhegan. The lexicon of sudoku [homepage on the internet]. c2005 [cited 2014 February 2]. Available from `http://www.playr.co.uk/sudoku/dictionary.php`.

[4] Éva Tardos Jon Kleinberg. *Algorithm Design*. Addison-Wesley, 2005.

[5] Johan Karlander. Föreläsning 7-8. c2008 [cited 2014 March 16]. Available from `http://www.csc.kth.se/utbildning/kth/kurser/DD2354/algokomp07/For0707+08.pdf`.

[6] Mária Ercsey-Ravasz; Zoltán Toroczkai. The chaos within sudoku. *Scientific Reports*, 2, October 2012.

[7] Johan Karlander. Föreläsning 9. c2008 [cited 2014 March 16]. Available from `http://www.csc.kth.se/utbildning/kth/kurser/DD2354/algokomp08/F_0809.pdf`.

[8] Donald Knuth. Dancing links. *Millenial Perspectives in Computer Science*, pages 187–214, 2000.

[9] Andrzej Kapanowski. Python for education: the exact cover problem. *CoRR abs/1307.7042*, October 2010.

[10] Radek Pelánek. Difficulty rating of sudoku puzzles by a computational model. 2011.

[11] Gaby Vanhegan. The rating system [homepage on the internet]. c2005 [cited 2014 April 10]. Available from `http://www.playr.co.uk/sudoku/ratings.php`.

[12] Dave Green. Conceptis sudoku difficulty levels explained [homepage on the internet]. c2006 [cited 2014 April 10]. Available from http://www.conceptispuzzles.com/index.aspx?uri=info/article/2#ID0EFC.

[13] The Python Software Foundation. The python wiki [homepage on the internet]. [updated 2014 January 24; cited 2014 March 20]. Available from http://wiki.python.org/moin/FrontPage.

[14] SciPy developers. Scipy [homepage on the internet]. [cited 2014 March 20]. Available from http://www.scipy.org.

[15] The Python Software Foundation. timeit — measure execution time of small code snippets [homepage on the internet]. [cited 2014 March 20]. Available from http://docs.python.org/2/library/timeit.html.

[16] The Python Software Foundation. time — time access and conversions [homepage on the internet]. [cited 2014 March 20]. Available from http://docs.python.org/2/library/time.html#time.clock.