2021

A study on the backtracking-based Sudoku solving algorithms

STEVE WEN
BSC OF COMPUTER SCIENCE
MCGILL UNIVERSITY
shuo.wen@mail.mcgill.ca

Overview

This study is mainly focused on the efficiency of backtracking-based algorithms solving 3x3 Sudoku puzzles. In the study, 3 backtracking algorithms and an additional dancing-links algorithm are implemented to carry out the study. The backtracking algorithm included has 3 types: intuitive, block preprocessing, row preprocessing. performance is rated on the combination of puzzle preprocessing generation time and the efficiency to cover the puzzle board. Instead of counting the elapsed time in the second part, the diverging branches of each set are counted. The testing data are all text files like Sudoku puzzles with difficulty ranging from very easy to very hard. After data collection, we then conclude DLX to be the most efficient one. However, there are still some limitations when it comes to higher-level solving cases. None of these algorithms could work effectively in 4x4 hard mode. Further research will focus on finding productive algorithms for handling high-level puzzles.

Acknowledgment

Special thanks to Professor Giulia Alberni for she authorized me to use the template.

This is for my parents, who always support me and love me.

Table of contents

1.Introduction

Sudoku is a puzzle game originating from France in the 19th century and got popularized in Japan in the nineteen-eighties. In the twenty years recently, the game has gained vast interest among folks. Competitions are held and new variants are created to meet the fast-growing interest in it. Meanwhile, scientists started to step in and talk about the mathematical and logical mechanisms behind the simple grid filling process. Rates of difficulty began to arise. More than that, with the rapid development of computational ability, computers started to play a significant role in finding a solution to an intricate puzzle. The ability to solve a puzzle in seconds stuns everyone as it would take minutes even hours for an experienced Sudoku player to solve it. This study will follow the footsteps of former researchers and take a step forward into analyzing the backtracking-based algorithms.

1.1 The Sudoku puzzles

Sudoku is a logical-based, combinatorial number-placement puzzle game. The classical Sudoku game is played on a big 9x9 grid made of 9 smaller 3x3 blocks. The major objective of such a game is to fill the board without breaking basic rules.
Three major rules to follow:

Row: Every row of the 9x9 grid should consist of 9 numbers exactly once from 1-9

Column: Every column of the 9x9 grid should consist of 9 numbers exactly once from 1-9

Block: Every block of the 9x9 grid should consist of 9 numbers exactly once from 1-9

These days, more and more variants of the Sudoku have been generated. Famous variants include chess Sudoku(Queen, Knight, King mode), and double-ten Sudoku. One thing to mention is that these additional rules are all based on the basic rules and can be viewed as an extension to the original one.

1.2 Solutions of the puzzle

A solution to the Sudoku puzzle is announced when the 9x9 grid has no empty cells and none of the three basic rules is broken. If additional rules apply, they have to be met simultaneously.
However, under certain conditions, a puzzle may have multiple solutions.

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

1.A A typical puzzle with two solutions

In 1.A above, it is clear to see that within the same block, either 2 or 7 can be put in the left first cell and the other in the second cell. More often than not, a puzzle would have a dozen solutions or more ready to be dug up. This seems to be of top difficulty to human players as they do the trial and error pretty slowly. However, this is when the computers come into play. With the astonishing power of machine computation, the deduction process can be done in a few milliseconds. This enables the full solution map to be found even when the puzzle is in its higher dimensions.

https://puzzling.stackexchange.com/questions/67789/examples-of-sudokus-with-two-solutions

2. Algorithm explained

Various algorithms have reached their tentacles into the Sudoku solving field. But the most intuitive and highly efficient method still gives its honor to the backtracking algorithm. This algorithm is closer to the strategy human players adopt to solve a Sudoku. Recall when faced with a Sudoku puzzle, human players will probably tentatively try a few numbers in certain spots and then expand to other empty cells. If he cannot maneuver, he would take a step back and start to try for other digits in that spot. After constant trial and error, and if the puzzle is manageable, a solution will luckily be found. Backtracking algorithms follow this idea and operate thousands of times to find a solution. Overview not granted, it might not get the right decision at certain points. This is where the machine falls behind experienced human players. They lack the overall view of the board to form a general decision. A false step will lead to a deep branch of backtracking which will inevitably fall into a seemingly infinite loop while the program is still trying hard to fill the grid in the wrong way. Therefore, it then becomes us programmers' work to pave the way for the machine. Backtracking algorithms indeed can be optimized to accelerate the process. This chapter will cover the methods that are optimized in the form of backtracking. intuitive implementation, row & block implementation, Dlx, a variant of backtracking, are included.

2.1 Backtracking algorithms

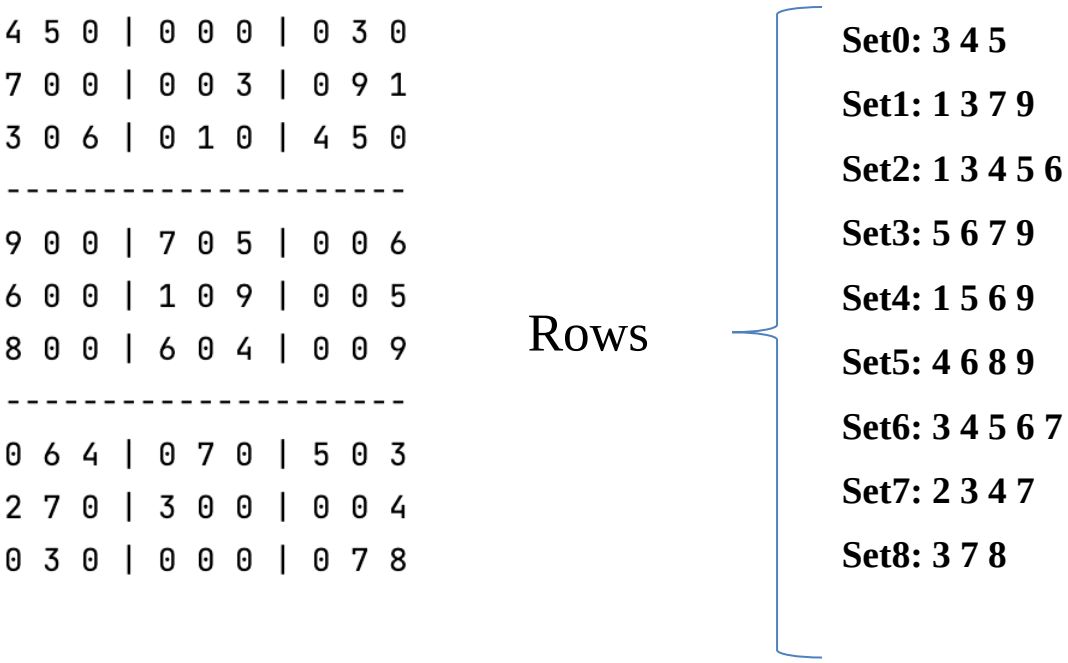
This part will talk about the Intuitive backtracking algorithm. Intuitive implementation follows the mindset of human beings, filling the empty cells one by one by trying for different digits. Block preprocessing will first find out all the possible arrangements for each 3x3 block based on the situation of the initial board. No actual numbers fill on the board. All the results will be stored in several ArrayLists consisting of integer arrays. These are called part solutions. Then backtracks based on these part solutions. Row preprocessing is almost the same as Block preprocessing except that it is to find the possible arrangements for the rows initially. This slight difference could make a huge variance in the final results.

2.1.1 Intuitive implementation

An intuitive backtracking algorithm acts like a human. Find an empty spot and try to fill it with some number. If it fits, then move on to the next cell and repeat the action until all the empty spots are cleared or none of the digits would fit. If it falls into the second case, the algorithm will then backtrack to the previous cell and try another digit in it and continue the backtracking process until the success or another failure.

All three backtracking algorithms have the additional advanced feature of storing the stage of the board into a series ArrayList of HashSets. There are 3 ArrayList standing for the three basic rules: Row's Rule, Column's Rule, and Block's Rule. Each of the ArrayList consists of 9 HashSet of Integers. For example, rows[0] to rows[8], represent the 9 rows on the board. Each row has a HashSet, which is actually in the ArrayList, to store the elements in the current row. The same case for the Columns and Blocks. This act is to increase the speed when checking whether the number to be filled into the board breaks one of these basic rules. If not, after the digit is put down onto the board, it will also find its place in the corresponding HashSet as well. When it fails in succeeding cells, the program backtracks and it will be deleted from the HashSet, meaning it is again an empty spot here. The mechanism behind is quite simple, HashSet could show whether the digit is in the corresponding row, column, block. In other words, it serves as the basic rule itself. Not being in the HashSet implies it could be a potential correct number.

```
ArrayList<HashSet<Integer>> rows
ArrayList<HashSet<Integer>> columns
ArrayList<HashSet<Integer>> squares
```

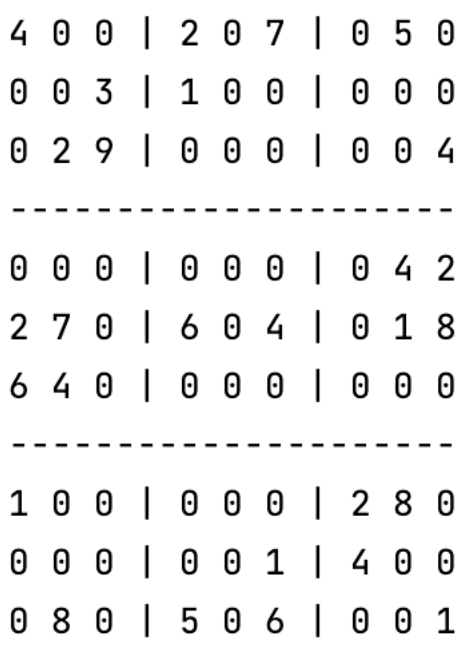


2. A An example showing how the Rows HashSet works

From the picture above, in the preprocessing time, every number in the grid was listed in its corresponding HashSet. A filling operation is made only when a digit is available in all three respective HashSets.
The algorithm stops when all the empty cells are drained, a success, or when possibilities die out, a failure.

2.1.2 Block-based implementation

Block-based implementation does a similar operation. The biggest distinction happens at the preprocessing time, the time when ArrayLists are built up. First, the program backtracks on every 3x3 block and gets all the possible solutions of the certain block. These possible solutions will be stored in an ArrayList of ArrayList each for further backtracking of the whole board. Then, Using the ArrayList, which is the collection of the possible solutions of every 3x3 block, the program will backtrack through all of them and try to find a complete solution. Ideally, this preprocessing will eliminate a large number of possibilities. Tests found out that the preprocessing takes only very little time. Although the preprocessing process would yield thousands of possible solutions, it operates on a rather small scale, which costs little time. The real time-consuming part is the latter one.



Block 0 part solutions

0 [4, 1, 8, 7, 6, 3, 5, 2, 9]

1 [4, 6, 1, 8, 5, 3, 7, 2, 9]

2 [4, 1, 6, 8, 5, 3, 7, 2, 9]

3 [4, 1, 8, 5, 6, 3, 7, 2, 9]

4 [4, 6, 1, 7, 5, 3, 8, 2, 9]

5 [4, 1, 6, 7, 5, 3, 8, 2, 9]

2. B A picture on the part solutions of Block 0

2.1.3 Row based implementation

The only difference between this and the previous one is that row implementation concentrates on the row part solutions instead of that of the blocks. However, the efficiency of these preprocessed algorithms cannot be told instantly. Tests show that row implementation shows a higher efficiency over Block one. This paper could not give precise mathematical proof of the reason behind it. But from a high-level view, more cells are exposed in the Row-based implementation, namely more constraints when performing the preprocessing. Also, this paper did not include the implementation of the column-based algorithm. Since the row-based one acts the same way as the column one, it could also be deemed as the transposition of column-based.

```
9 1 3 | 6 0 0 | 0 0 0
2 5 0 | 0 8 7 | 0 9 0
0 0 0 | 0 0 0 | 0 4 0
---------------------
5 8 0 | 0 4 6 | 0 0 7
0 0 0 | 0 5 0 | 0 0 0
7 0 0 | 8 9 0 | 0 5 6
---------------------
0 9 0 | 0 0 0 | 0 0 0
0 6 0 | 3 7 0 | 0 2 9
0 0 0 | 0 0 5 | 3 6 4
```

Row 5 part solutions

0 [7, 3, 4, 8, 9, 2, 1, 5, 6]

1 [7, 4, 2, 8, 9, 3, 1, 5, 6]

2 [7, 2, 4, 8, 9, 3, 1, 5, 6]

3 [7, 3, 4, 8, 9, 1, 2, 5, 6]

4 [7, 4, 1, 8, 9, 3, 2, 5, 6]

5 [7, 3, 2, 8, 9, 1, 4, 5, 6]

6 [7, 3, 1, 8, 9, 2, 4, 5, 6]

7 [7, 2, 1, 8, 9, 3, 4, 5, 6]

2. C A picture on the part solutions of Row 5

2.2 Dancing Links algorithm

The basic idea of the Dancing Links algorithm is quite simple. But its applications are not.

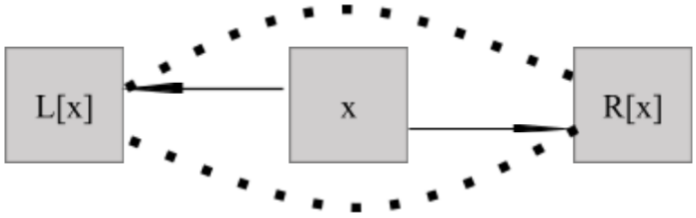
The removal of a Node from a Doubly Linked List takes 2 steps, and so as the recovery of it. Professor Knuth said this idea needs to be better known as it might prominently be powerful in certain recursive algorithms.

$$L[R[x]] \leftarrow L[x], \qquad L[R[x]] \leftarrow x$$
$$R[L[x]] \leftarrow R[x], \qquad R[L[x]] \leftarrow x$$

(x to be the Node in the Doubly Linked List)

This seems to be obvious but not familiar to people. After the disconnecting operations of the first column, Node x still possesses the reference to its left and right Node. But the left Node of x directly points to the right Node of x, pointing over x, and vice versa.

One thing to bear in mind is that the Doubly Linked List has to be circular. This setup guarantees that the Head Node and Tail Node still connect when all elements between are moved out.



The biggest problem of the implementation of DLX (Dancing Links X algorithm) is not the DL itself but the transformation from the Sudoku puzzle to the exact cover problem.

The exact cover problem is to find a cover from all subsets of some collection where subsets combine to make a complete set, the cover. There is no overlap between the subsets. A simple example from WiKi follows.

Let S = {*N*, *O*, *P*, *E*} be a collection of subsets of a set *X* = {1, 2, 3, 4} such that:

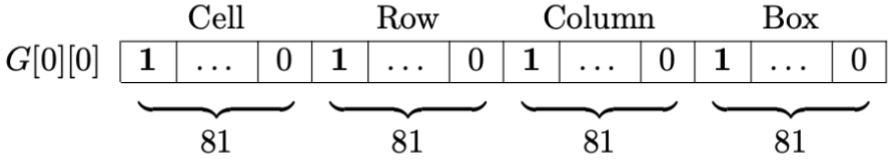
***N* = {}, *O* = {1, 3}, *P* = {2, 3}, *E* = {2, 4}.**

The subcollection {*O*, *E*} is an exact cover of *X*, since the subsets *O* = {1, 3} and *E* = {2, 4} are disjoint and their union is *X* = {1, 2, 3, 4}.

The subcollection {*N*, *O*, *E*} is also an exact cover of *X*. Including the empty set *N* = {} makes no difference, as it is disjoint with all subsets and does not change the union. [1]

The puzzle matrix needs to be transformed into a cover matrix consisting only of 0 and 1s.

Four constraints are applied to make the cover happen. Cell Constraint, Row Constraint, Column Constraint, Box Constraint are combined to cover the board. Every constraint has 81 columns in it since there are 81 cells in a 3x3 Sudoku puzzle game.



2. D An illustration on the first line of the cover matrix

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

2. E The first 81 columns of the Cell Constraint
(1- 81 refers to the 0- 80 columns)

	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**

2. E The second 81 columns of the Cell Constraint
(1- 81 refers to the 81- 161 columns)

	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
⋮													

2. F The second 81 columns of the Column Constraint
(1- 81 refers to the 162- 242 columns)

	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
⋮										

2. G The second 81 columns of the Cell Constraint
(1- 81 refers to the 243- 323 columns)

A solution is found when all the columns of the cover matrix have exactly only a 1 in it.

Then certain data structures and functions are operated to display the results.

The basic setup of the program follows that of the paper from Matias Harrysson and Hjalmar Laestander. Major three methods are as follows.

Search: This function undertakes the main task of the whole algorithm. It performs the integration of cover and uncover. **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** = [].

cover: the cover operation that removes c from the doubly linked list of column objects and removes all data objects under c.

uncover: The reverse function of cover. Using the technique of Dancing Links.

```
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
```

```
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]
```

```
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
```

2. H A set of illustrations on the major functions o

3. Method

Java is chosen as the programming language. Although C was significantly faster and was more popular than Java, Java has advantages over C. The biggest one is that Java is Object-Oriented. This is particularly useful since in the DLX part, both Column & Node objects are used to process the board. Without the support of OOP, this is hard to achieve. Secondly, Java has more installed classes that help to perform some basic functionalities, while in C it doesn't. Time has been greatly saved as we do not need to build tools on our own. Thirdly, garbage collection is automatically done here. We need to cover the Sudoku board over and over. Some steps may not perform completely or malfunction and make irreversible results. With Java, some confusing outcomes could be perfectly avoided.

3.1 Setup of the test

All the code was running on the same hardware and the same software to maintain consistency in the results.

3.1.1 Hardware

Model Name:	MacBook Pro

Model Identifier:		MacBookPro11,4
Processor Name:		Quad-Core Intel Core i7
Processor Speed:		2.2 GHz
Number of Processors:	1	
Total Number of Cores:	4	
Memory:		16 GB

3.1.2 Software

IDE platform was used to simplify the research process.

IntelliJ IDEA 2021.1.3 (Ultimate Edition)
Build #IU-211.7628.21
Runtime version: 11.0.11+9-b1341.60 x86_64
VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o. macOS 11.4
GC: G1 Young Generation, G1 Old Generation
Memory: 750M
Cores: 8

3.2 Comparison

Several ways exist to measure the efficiency of Sudoku solving algorithms. In some cases, the elapsed time is taken to tell the speed, while others like Professor Knuth used branches of the recursion to figure out the capability. In our case, both are utilized as the process is complicated to tell instantly.
For the puzzle difficulty rating, we did not find any quantitative method measuring it.

3.2.1 Puzzle difficulty comparison

The rate of the difficulty on a puzzle is hard to be measured using quantitative methods, only qualitative methods define it ambiguously. Criteria vary, but one common sense has formed is that number of clues does not define the difficulty completely. Some say that time taken to solve the Sudoku on average should be the ruler, while others claim that rank is set according to the difficulty of the techniques required to generate a solution.

For this research, we did not bother to define the difficulty rigorously because our main purpose is to compare the efficiency. We just followed the rate on the puzzles set by other professional Sudoku solvers.
The sources of these Sudoku puzzles are not restricted, this might cause some deviation in the outcome.
Three levels of difficulty are set: Easy, Medium, Hard.

3.2.2 Preprocessing time comparison

Preprocessing is an important part of this research. It is capable of reducing a great quantity of overall time, namely increasing efficiency. This can be directly measured by using the class System. How I implemented it follows.

```
long start = System.currentTimeMillis();
preprocessing();//preprocessing
long end = System.currentTimeMillis();
long time = end- start;
```

All the methods that undertake the task of preprocessing are packaged into one method so that the runtime could be measured once.

3.2.3 Solving efficiency comparison

For this part, we adopt the method from Professor Knuth of counting the branches instead of time. Time is not a perfect tool here. Different methods will have different operations on the board or other boards. Fetching, or getting access to some data, doing some calculations will all be counted in. This variance may fail to fully represent the true power of the algorithm. On the contrary, branching counting seems to be a nice path. This type is independent of other operations, only counting the sprawl of the branches. However, The implementation of branching counting is easier said to be done. Counting must happen at the deepest end of each backtracking. Otherwise, some parts will be counted repeatedly, causing an over-count.
Limited places are for the branch counting. One is where the method returns true when a solution is found. Another one happens when a certain Spot fails all the possible numbers.
An auto counter is placed to record those branches.

3.3 Find multiple solutions

The feature of finding multiple solutions also comes into play as some simple cases would have more than one solution.
This feature is achieved by returning false when successful.

```
if(fill()){
        if(!allSolutions) return true;
        solutions.add(currentSudokuObject);
        return false;
```

fill is the Boolean method that undertakes the general task of doing backtracking.
allSolutions is the Boolean option, whether to find all the solutions of a certain Sudoku object.
When fill returns true, it means the solution is valid. Return false makes sure to be in the loop backtracking for other solutions.

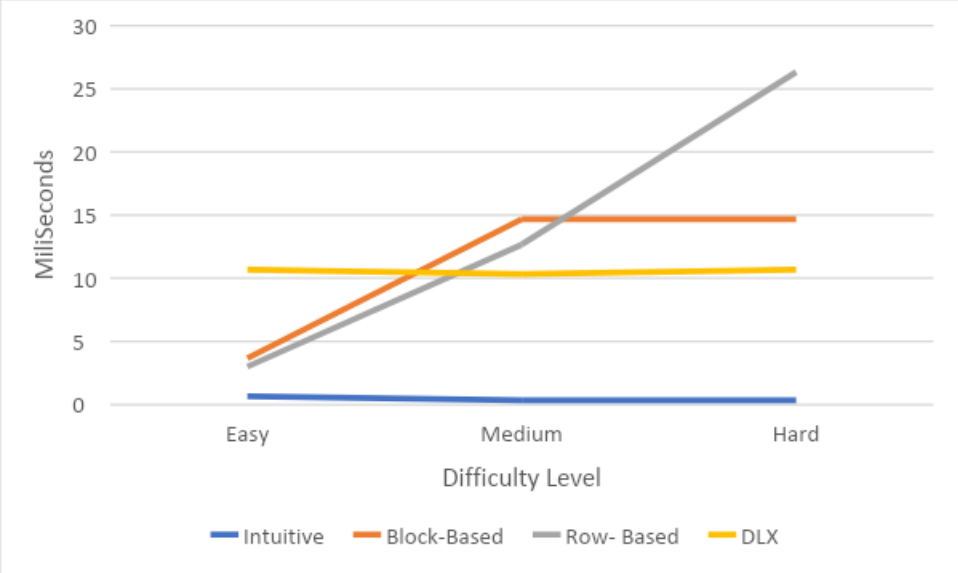
3.4 Constraints of algorithms

The constraints in these algorithms are visible. In our tests, none of these algorithms could survive after the medium level of 4x4. All of them would get stuck in some middle points where recursion branches become massive and uncontrollable. This shows that our algorithm is not efficient enough in proficiently reducing the branches of backtracking. Further optimizations need to be done to solve this.
Another constraint is that these algorithms lack the necessary overview of the board. Video Footages show that experienced human players have some sense of the filling trend. This is particularly complicated to achieve in a program. More studies need to be done on this part.

4. Analysis

Analyses are separated into two parts, Preprocessing time distribution & Diverging branches counting. This could not be wrapped into a whole as the results would differ under certain circumstances. Separating them apart helps to present the outcome much more clearly.
DLX proves phenomenally outstanding in reducing branches, while it might take some time in the preprocessing.

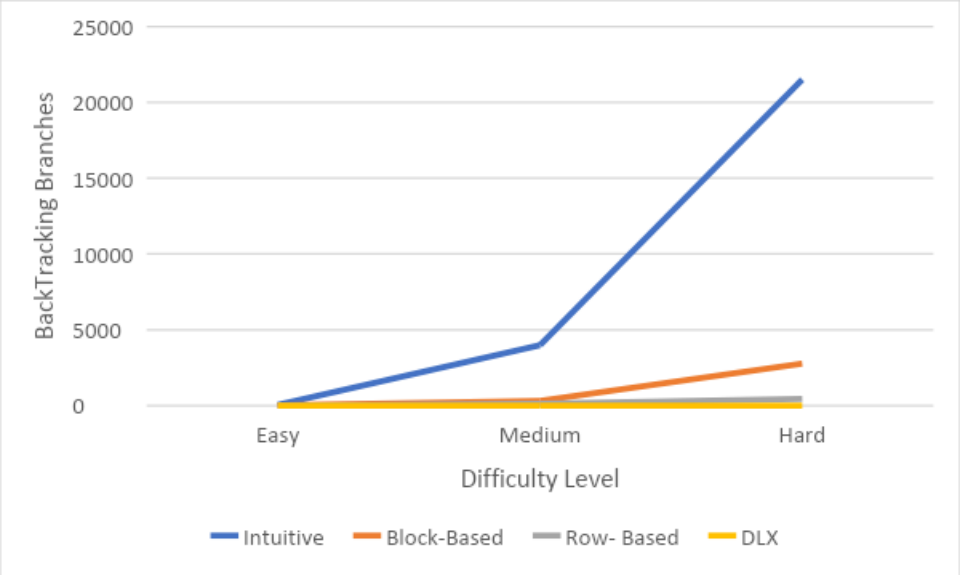
4.1 Preprocessing time distribution



4.A Time distribution for the preprocessing process

The intuitive algorithm and DLX form nearly a straight line across three sets as their preparation times are steadily the same, with DLX more work to be done, taking a bit longer. The results of the first two levels for Block-Based and Row-Based algorithms are as expected. The block one should consume more time than the Row one as the latter has more constraints on them, reducing potential solutions.
Nevertheless, the preprocessing time of the Block-Based Algorithm did not elevate as expected when it comes to the Hard mode. Further research needs to tell whether this was a coincidence or the testing puzzle was not hard enough or if the algorithm is powerful enough in hard mode.

4.2 Diverging branches counting



4.B Counting for the backtracking recursive branches

From the line chart, it is clear to see that the dancing links algorithm (DLX) performs incredibly amazingly as well as the Row-Based preprocessing backtracking algorithm does. Block-Based preprocessing algorithm went steady in the first two levels but has a surge in the hard mode. The intuitive implementation plays the worst, with an amount of over twenty thousand branches.

This could be explained qualitatively. The intuitive one follows no special rules and just tries to fill every cell from 1 to 9. This might lead to deep recursive branches as the program would find dead ends near the very end of the puzzle. While the Block-Based algorithm reduces the branches much as it has to preprocess before the backtracking on the board. All the blocks would have many potential part solutions. With all of them selected and combined, a whole solution is easier to be found. The Row-Based algorithm follows the same idea as the Row-Based algorithm. The difference has one possible explanation: more constraints are considered in the Row-Based one when forming the part solutions. Thus, it will generate fewer potential part solutions for each row, leading to fewer branches. The performance of DLX is phenomenal. It only creates one

backtracking branch for most testing cases. The case could be explained as it has the most constraints in backtracking. It has a different path from the other three. Stricter cover boards narrow the diverging branches.

Few branches do not ensure it costs little time to run the program. In the real test set, DLX often takes a long time to cover and fill the board. One possible reason for that is the algorithm is always dealing with Doubly-Linked lists and objects. However, in higher levels, DLX might make huge leaps in running time compared to others because basic algorithms can be easily trapped into the deep backtracking branches.

5. Conclusion and Further thoughts

Dancing Links X Algorithm proves to be the best algorithm to solve the Sudoku puzzle. It has outperforming power to effectively reduce diverging branches. Unlike other algorithms, DLX does not sprawl and will get to the correct answer right away. In the area of preprocessing, DLX takes almost the same amount of time.

Although an intuitive backtracking algorithm takes no time in preprocessing, its running process will need many more bunches of diverging branches as it will try one number after one. This would lead it to be stuck somewhere in the middle of the backtracking where branches surge and become massive.

The row-based algorithm seems to be more effective than the Block based one in branch backtracking. Rigorous mathematical proof could not be given as we lack the necessary skills on it. Further research needs to land on the proof of it.

In the higher dimensions, none of these algorithms prove to be efficient. Even DLX will stop its way of finding the right answer. In the future, we have to find some algorithms that can solve those 4x4 hard puzzles plus.

6. Bibliography

[1]. David Austin Puzzling Over Exact Cover Problems. [homepage on the Internet]. 2009 [Cited Jun.18. 2021]. Retrieved from: American Mathematical Society, Website: http://www.ams.org/publicoutreach/feature-column/fcarc-kanoodle

[2]. HARRYSSON, M., & LAESTANDER, H. Solving Sudoku efficiently with Dancing Links (Dissertation). [homepage on the Internet]. 2014 [Cited Jun.21. 2021]. Retrieved from: KTH Royal Institute of Technology, http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-157551

[3]. Donald E. Knuth The art of computer programming volume 4 pre-fascicle 5c. [homepage on the Internet]. 2019 [Cited Jun.21. 2021]. Retrieved from: Instituto de Informática https://www.inf.ufrgs.br/~mrpritt/lib/exe/fetch.php?media=inf5504:7.2.2.1-dancing_links.pdf

[4]. Jonathan Chu A Sudoku Solver in Java implementing Knuth's Dancing Links Algorithm. [homepage on the Internet]. 2006 [Cited Jun.26. 2021]. Retrieved from: https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/sudoku.paper.html

[5]. Gieseanw Solving Sudoku, Revisited [homepage on the Internet]. 2011 [Cited Jun.26. 2021]. Retrieved from: WordPress, https://gieseanw.wordpress.com/2011/06/16/solving-sudoku-revisited/