KTH Computer Science
and Communication

Performance and Scalability of Sudoku Solvers

Prestanda och Skalbarhet för Sudoku Lösare

VIKSTÉN, HENRIK
NORRA STATIONSGATAN 99
STOCKHOLM 113 64
073 643 76 29
HVIKSTEN@KTH.SE

MATTSSON, VIKTOR
FURUSUNDSGATAN 10
STOCKHOLM 115 37
073 061 74 90
VIKMAT@KTH.SE

Bachelor's Thesis at NADA
Supervisor: Mosavat, Vahid
Examiner: Björkman, Mårten

TRITA xxx yyyy-nn

Abstract

This paper aims to clarify the differences in algorithm design with a goal of solving sudoku. A few algorithms were chosen suitable to the problem, yet different from one another. To take an everyday puzzle and utilize common computer science algorithms and learn more about them. Get relevant data to see how they perform in situations, how easily they can be modified and used in larger sudokus. Also how their performance scales when the puzzle grows larger.
From the results Dancing links was the fastest and scaled best of the algorithms tested, while Brute-force and Simulated annealing struggled keeping consistent results.

Referat

Prestanda och Skalbarhet för Sudoku Lösare

Detta dokument syftar till att klargöra skillnaderna i algoritmer med målet att lösa sudoku. Välja ett par olika algorithmer lämpliga för problemet, men som samtidigt skiljer sig från varandra. Att ta ett dagligt pussel och utnyttja vanligt förekommande algoritmer inom datavetenskap och lära sig mer om dem. Få relevant data och se hur de presterar i olika situationer, hur lätt de kan modifieras och användas i större Sudokus. Även hur deras prestanda skalar när pusslet blir större.
Dancing links var den snabbaste algoritmen och skalade bäst av de som testades. Brute-force och Simulated annealing inte var lika konsekventa genom alla tester och betydligt långsammare i överlag.

Contents

Chapter 1

Introduction

Sudoku is a game traditionally played with pen and paper, but can be solved a lot faster with computer algorithms. Computers can solve a 9x9 sudoku with roughly 20-25 given numbers within less than a second, way faster than a human ever could. These algorithms open up a wide range of design paradigms and ways to solve the puzzle. Everything from brute forcing, backtrack to stochastic search. In this report different algorithms will be tested and conclusions about performance in different scenarios and sizes will be made.

1.1 Purpose

In computer science, algorithms are the foundation of complex problem solving. In this report focus is on solving sudokus.
This report is intended to reach an understanding in the scalability and performance of different sudoku algorithms. What design paradigms fits the best for a specific situation, how the performance scales over various sizes and the major differences between designs. But also discuss the basics of sudoku solving.

1.2 Problem Statement

There are several different algorithms for solving sudoku, different programming languages and design paradigms. This report will take a few of these methods, analyze them and present the data and discuss it. Sudoku can be considered quite an easy puzzle to solve but the algorithms can be complex. The main goal is to see how different design paradigms perform and scale when the task get tougher, in the end finding the best algorithm and their weaknesses.

- How does the algorithms perform?
- How do the algorithms scale when the puzzles grow?
- How is the algorithm used to solve sudoku?

- Which algorithm works best for sudoku and why?

1.3 Scope

Limiting the selection of algorithms and testing is crucial. At the same time extract reliable data in the testing environment. That is why this report only focuses on Java and the following algorithms:

- Brute-force
- Simulated annealing
- Dancing Links

Focus is on measuring the time an algorithm takes to solve 52 amount of sudoku puzzles and comparing different implementations and sizes of puzzles. It was a conscious choice to only test 52 boards at a time.

1.4 Statement of Collaboration

This project is intended to be a group effort and it is a clear success, both members have been actively taking part in all aspects of the projects. Every single detail has been discussed and close teamwork has always been our strong side. Although, some work tasks have been divided slightly, while Viktor wrote our testing interface Henrik searched the web for information, algorithms and started writing. In the testing phase both of us were involved checking if the numbers were feasible. When testing was done, both of us continued working with the remainder of the document.

1.5 Summary of Results

Considering the scope of the test and that algorithms can be implemented in different ways results in this paper might not always be accurate, however, results with our methods and algorithms are all extensively tested. Discussion and preferences are based on data and personal observations during the tests. We do not think that these are the best algorithms ever implemented for these design paradigms but decided to use easy to find algorithms that came from a somewhat recognizable source.

1.6 Definitions

Box a 3x3 part of the sudoku puzzle.

Clue/Givens a number already in the sudoku puzzle that cannot be changed.

Grid/Board the entire sudoku puzzle.

Cell a 1x1 part of the sudoku puzzle.

1.7 Overview of the Document

Chapter 2 Background, introduction to sudoku and algorithms explained.

Chapter 3 Approach, testing enviroment and interface descriptions.

Chapter 4 Results, data from tests.

Chapter 5 Dicussion, discussing the results and algorithms.

Chapter 6 Conclusion, what this paper achieved.

Chapter 7 References, a list of sources of information used for the paper.

Appendix A collection of raw data from tests, including tables and diagrams.

Chapter 2

Background

2.1 What is Sudoku?

Sudoku is a logical number puzzle. The conventional puzzle consist of a 9x9 grid of smaller squares divided into nine 3x3 grids. These squares is to be filled with a number ranging from one to nine. Some numbers are given, how many you get and positioning alters the difficulty of the puzzle.

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

Figure 2.1. Unsolved sudoku, 28 givens.

2.1.1 Rules of Sudoku

In order to solve a sudoku puzzle you must follow these rules.

Rule 1 *A number may only appear once in every row.*
Using the board in fig. 2.1 we notice that following rule one, “5 1 2 | 3 4 6 | 8

7 9" could be a possible solution for the first row. However this brings us to the second rule.

Rule 2 *A number may only appear once in every column.*
Using rule two we notice that previous solution does not work. In column two row five we have a set number of one which mean that it is not possible for us to set this number in the second column of the first row. There are even more errors but considering them we could try shuffle the numbers around to prevent the errors. We notice that "5 6 7 | 4 3 1 | 8 2 9 " fullfill rule one and two.

Rule 3 *A number may only appear once in every sub 3x3 grid.*
Rule three does once again make the solution invalid. Using this rule we notice that the numbers in column 2 and 6 are wrong.
The first 3x3 subgrid contains the predetermined numbers five and six. This mean that placing another six in this subgrid would be against the rules. Once again moving around the numbers, this gives us a solution "5 2 1 | 4 3 7 | 8 6 9" and we notice that all three rules are fulfilled and can continue with the rest of the board resulting in a complete solution such as the one shown in fig. 2.2.

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

Figure 2.2. Solution of fig. 2.1

The board does not have to be solved one row at a time but can be solved in any manner following the rules.

2.2 Solving Sudoku by Hand

Sudoku solving can be seen as a combinatorics problem. This helps us determine the number of possible solutions. Although this does not help a normal person solving sudoku. To be able to solve a harder sudoku like the one from Le Monde magazine (fig. 2.3) you would have to use special techniques.

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

Figure 2.3. Hard sudoku from Le Monde magazine

There are several commonly known techniques for solving Sudoku by hand, however, all of them are limited by the human brain. According to miller's law the average human brain can only hold 7+/-2 things in the working memory [3]. This means it is nearly impossible for a human to remember all cells in which a specific number or numbers can be placed. That is why it is very common to take notes, write possible numbers for a specific cell within it. This helps solving the puzzle and eventually eliminates candidates.
That is a good start, although all puzzles are not that simple. Like the one in figure 2.3, we might have to implement another strategy to solve it. There are a couple of advanced sudoku solving techniques, but among the most common are X-Wings [4] and Swordfish [5].

2.3 Solving Sudoku Using a Computer

A lot different from a human, computers are not limited by working memory in the same way. The ability to save and retrieve data from memory makes a computer the perfect sudoku solver. It excels at working tasks that are repetitive, logical and does it fast. For solving sudoku there are a few commonly used algorithms, while

Brute-force is the most intuitive way for a human to write code it might not be the best for a computer to solve it.

2.3.1 Backtracking

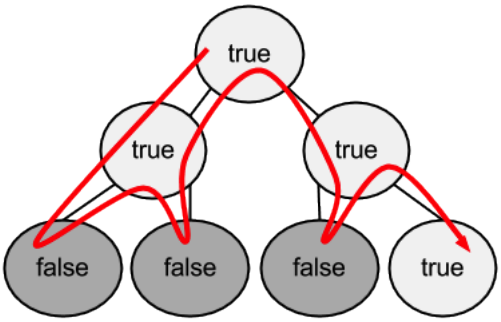


Figure 2.4. Sample backtracking tree

Backtracking is a commonly used constraint satisfaction problem solving algorithm[10]. Backtracking is a form of search that climbs recursively down a tree made of different choices. Whenever a backtracking algorithm is used its goal is to reach the very end of its recursion tree where the solutions are but in order to reach the end you will have to search your way down along a valid path. Fig 2.4 shows the path the algorithm will work in its worst case along a small binary tree.
The algorithm will start at the top by checking if its set of constraints hold and decide that it does (true) and decide to continue by recurse downward one level. In this spot it once again check if the constraints hold and continue downwards as it does. But when reaching the third level it notice that the constraints no longer holds (false), then we know that the rest of that branch is also false and does not have to check the rest. The algorithm will have to use the special functionality of backtracking and go up one level. As the algorithm go up a level in the recursion tree it will now know that the previously visited node is false. This allows it to go down a new path if available, or up another level if there is no other path. Creating a form of brute-force search through the solutions.
It is important to understand that a backtracking algorithm is by no means a true brute-force algorithm. If the binary tree in fig 2.4 would have been one level deeper a backtracking algorithm would not have to check the majority of the fourth level as the solution does not hold in level three, while a true brute-force algorithm would have. Thous backtracking is usually faster than a brute-force solution.

2.3.2 Brute-force Search

Brute-force is probably the most common way to solve a sudoku using a computer. However a true brute-force solution for sudoku is not feasible due to the fact that

there are approximately $6.671x10^{21}$ different 9x9 sudoku boards [8]. Therefore the brute-force approach to sudoku is actually a backtracking algorithm that is very natural for humans and consists of a few, very simple steps for solving.

1. For each possible number.
2. Check if any number is valid for the cell.
3. If a number is valid, add it to the cell and step to next cell else backtrack to previous cell.
4. If a number was added to the last cell the sudoku has been solved.

While the brute-force approach is simple to understand it might not be the most efficient one. As numbers are added as soon as they see fit they may very well create problems for future cells creating a need to backtrack. In the worst case, some of the very first cells would be wrong and you will have to backtrack the entire board which means redoing all of the solution.

2.3.3 Simulated Annealing

Simulated annealing is a probabilistic method for solving the global optimization problem [6]. The name originates from the metallurgy method annealing where you grant metals different properties by heating and cooling it. Simulated annealing works in a similar way [13].
To be able to utilize this method for sudoku we would have to define what has to be optimized by the algorithm. As a sudoku is considered solved when all the cells are filled and every rule is applicable to each cell, we can define this as the number of cells violating the rules. When the number of cells that contradict the rules reach zero, a completely solved board has been achieved.
Solving a sudoku using simulated annealing works by creating a potential solution to the problem by filling all unfilled cells with a randomized value. With the entire board filled, the algorithm will count how many errors that has been made and iteratively swap two cells of these boxes updating the solution. If the new solution contains less errors than the previous it is considered better and will be kept as the starting point for future iterations. This is the cooling part of annealing. Though as a sudoku board can be almost completely solved while still being entirely wrong, it is also necessary to save a few solutions that is considered worse. This is the heating part of annealing and is a necessary part of simulated annealing. Without being able to reheat a solution, the algorithm will get stuck at trying to solve the almost solved board even though it is unsolvable.

2.3.4 Dancing Links

Dancing links is an algorithm that utilizes a pre calculated sparse matrix of constraints to solve the exact cover problem efficiently. Exact cover is a constraint programming problem where each constraint should be fulfilled once and only once.

As Helmut Simonis states, Sudoku is a typical constraint programming problem [1]. Constraint programming differ from usual programming problems as you do not specify steps for a program to do in order to reach a solution. Instead you create a set of conditions or constraints that the solutions share. Constraints can tell what the solution should be or what it should not be, allowing the program to easily filter out faulty solutions [14].
For sudoku the constraints to be fulfilled are:

- A cell constraint: Each cell can only be filled with one number.
- A row constraint: A number may only occur once in each row.
- A column constraint: A number may only occur once in each column.
- A box constraint: A number may only occur once in each box [9].

This will create a sparse matrix of size (Sudoku size)^3 x 4*(Sudoku size)^2 where one row for a 4x4 sudoku could look like this:

0000000000000100 0000000000000001 0000000100000000 0000000000010000

where the first 16 numbers state the cell constraint, second 16 numbers the row constraints, third column the constraints and forth the box constraints. This particular line could be read as “Cell 14 is filled. Row 4, column 2 and box 3 has been occupied by number 4".
Dancing links uses this matrix to solve the sudoku board using exact cover with the very simple idea that you can both remove and add an element to a double linked list just by moving its nodes next and previous pointers [12]. Donald Knuth, creator of the dancing links algorithm described the algorithm by using a few short examples and so shall this paper also use a small example. Note that this example is based on a lecture from Colorado State University[11].
Consider you have the sparse matrix

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

Creating this sparse matrix using double linked list for its rows and columns where each 1 is represented by a node with up/down pointers for columns and left/right for rows would create a data structure as shown in fig 2.5. It is important to keep column head nodes(a-g) to easily know which columns that has been removed and a primary head node(h) to determine when a solution has been found.

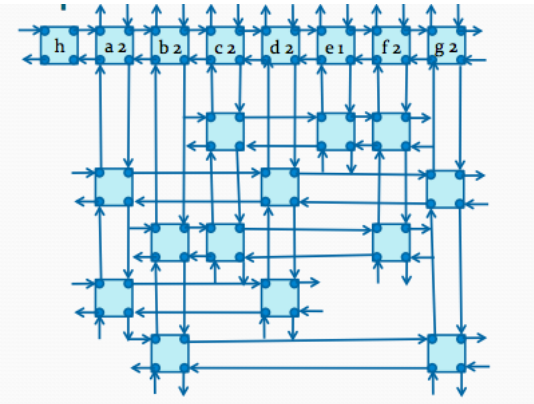


Figure 2.5. Linked list structure of a sparse matrix.

Dancing links uses this data structure to solve the exact cover problem using the benefit that it can easily remove a constraint and all the related constraints. For example, if we were to decide that column a, row 2 holds for our solution we would have to remove this column and row from the matrix and put it into our solution. This can easily be done by moving neighboring nodes next and previous pointers around the nodes of this row and column as shown in figure 2.6.

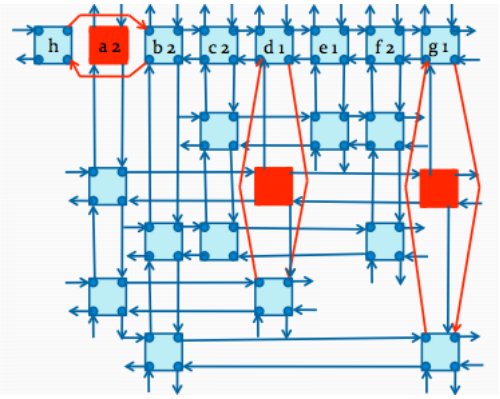


Figure 2.6. fig. 2.5 with column a, row 2 covered.

Now, to solve the exact cover problem we can not have more than one of each constraint fulfilled. We would have to remove all rows that share these constraints. Easily enough, these rows are the neighbors of the nodes that were just removed

and therefore, all rows that contradict the exact cover solution can be iteratively removed.
Once this has been done the columns a, d and g are considered “covered” and they will no longer be a part of the matrix along with the rows that share these columns. The matrix would now look like this.

$$\begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{matrix}$$

However, as columns b, c, e and f are still uncovered and there is no way to choose one of these rows without covering the other, it is impossible to find a solution using column a, row 2 as a starting point. This mean theres a need to backtrack and redo the choice of starting point. In this example the solution would be to cover in the order:

1. Column a, row 4.
2. Column b, row 3.
3. Column c, row 1.

This can easily be checked by hand. Adding the rows together and check if it ends up in a row consisting of only ones but for dancing links it is when the primary head node links to itself.

Chapter 3

Approach

To be able to produce reliable results that are proportional to each other it was early decided to make sure the algorithms had the same potential within the testing software. As stated in the scope it was also decided to use Java as the programming language for all of the tests. The main concern was not getting reliable data measured in wall-clock time, but rather get data that can be compared between design paradigms.

3.1 Testing Enviroment

3.1.1 Hardware and Software

To ensure that the test data was not affected by external conditions the same computer was used for all the tests, and they were all done at the same time. The computer specifications can be found below.

Model Lenovo Y500 laptop

CPU Intel Core i7 3630QM 2,4 GHz

Memory 8GB SO-DIMM DDR3 @ 1600MHz

GPU NVIDIA GeForce GT 650M SLI

HDD 1000GB Storage, 16GB SSD, Hybrid, 5400 RPM

The tests ran on Windows 8 with Intellij Idea version 12.0.4.

3.1.2 Testing Interface

To make the testing easier we decided to write a small testing interface. The testing interface is very simple. It serves four purposes.

1. To load sudoku boards from file or call a sudoku generator to create a set of sudoku boards.

2. To safely copy sudokus for each algorithm in order to prevent that a sudoku board is solved when another algorithm is being tested.
3. To store and present the time an algorithm was working.
4. To present runtimes and average runtime.

3.2 Time Measurement

The time measurement of the tests will be run as follows:

Initialize Algorithm *a*.

Store current system nanotime.

Solve the sudoku board using algorithm *a*.

Return difference in system time since before solving.

It was decided that the algorithms should be initialized before starting time measurements to show actual solving time instead of time necessary to set up the algorithm. This mean that algorithms such as dancing links that require certain one-time pre calculation will get a slight advantage.
The best and worst time will not be included in the average time in order to grant more precise results due to caching and other factors.

3.2.1 Nanotime

Due to the nature of certain algorithms we have chosen to measure times using java's nanoTime() method in the system library. There are several reasons for this. Java's API writes this about nanoTime:
"Returns the current value of the most precise available system timer, in nanoseconds. This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed but arbitrary time (perhaps in the future, so values may be negative). This method provides nanosecond precision, but not necessarily nanosecond accuracy. No guarantees are made about how frequently values change. Differences in successive calls that span greater than approximately 292 years (263 nanoseconds) will not accurately compute elapsed time due to numerical overflow." (Copied from java API) [7].
As we only want to know the elapsed time for the algorithms and not wall-clock time, the API state that we will get a time in nanoseconds with nanosecond precision but not nanosecond accuracy. While a good accuracy would have been prefered we still think that nanoTimes accuracy is close to other time methods such as currentTimeMillis() due to the reason that the algorithms are so fast in most cases that currentTimeMillis() would return a time of zero as it does not return a floating point value. The great amount of zeros would reduce the accuracy perhaps even

more than what the nanoTime method does.
When data is presented every reference to nanoseconds is Javas nanoTime function, not wall-clock time.

Chapter 4

Results

4.1 4x4 Sudoku

The 4x4 test results are presented below. Notice that in the report only the average times are shown but, additional data and diagrams can be found in Appendix A.

4.1.1 52 Random boards

Brute-force average: 48220ns
Simulated annealing average: 1339234ns
Dancing links average: 14582ns

4.1.2 52 Set boards

Brute-force average: 54087ns
Simulated annealing average: 618138ns
Dancing links average: 13205ns

4.1.3 10 Boards with no clues

Brute-force average: 107871ns
Simulated annealing average: 2375996ns
Dancing links average: 48536ns

4.2 9x9 Sudoku

The 9x9 test results are presented below. Notice that in the report only the average times are shown but, additional data and diagrams can be found in Appendix B.

4.2.1 52 Random boards

Brute-force average: 138742ns
Simulated annealing average: 8166863ns
Dancing links average: 47672ns

4.2.2 52 Set boards

Brute-force average: 474641ns
Simulated annealing average: 6519226ns
Dancing links average: 43755ns

4.2.3 10 Boards with no clues

Brute-force: 1048722ns
Simulated annealing: 33935818ns
Dancing links: 237391ns

4.3 16x16 Sudoku

The 16x16 test results are presented below. Notice that in the report only the average times are shown but, additional data and diagrams can be found in Appendix C.
Simulated Annealing has been removed for this test due to being too slow. Attempts at completely solving a board were unsuccessful even when allowing 100000 iterations to be done, the time would not give us any useful information.

4.3.1 52 Random boards

Brute-force average: 741947ns
Dancing links average: 155394ns

4.3.2 52 Set boards

Brute-force average: 4087887ns
Dancing links average: 171438ns

4.3.3 10 Boards with no clues

Brute-force average: 17664554ns
Dancing links average: 1802376ns

4.4 Scaling

A 4x4 sudoku has 16 cells to be filled, a 9x9 has 81 and a 16x16 has 256. This mean the problem increase by about 5 times when the problem increase to a 9x9 sudoku from a 4x4 and about 3 times when it grows from 9x9 to 16x16.

4.4.1 Brute-force

Average solve time increased with 7.9 times when problem increased from 4x4 to 9x9. Following, when the problem grew from 9x9 to 16x16 the average solve time increased with 13.5 times.

4.4.2 Simulated Annealing

Average solve time increased with 11.2 times when the problem increased from 4x4 to 9x9.

4.4.3 Dancing Links

Average solve time increased with 4.3 time when the problem increased from 4x4 to 9x9. Following, when the problem grew from 9x9 to 16x16 the average solve time increased with 6.5 times.

Chapter 5

Discussion

Solving sudoku is no problem for a computer. It is not a matter of minutes, it is a matter of seconds or even less. Our tests are not measured in real clock time, however, that is because solving a single sudoku puzzle with javas millisecond function returns the answer 0 too often. For a human wanting to solve a normal 9x9 sudoku with a computer it does not matter what algorithm you choose. But to achieve top speeds certain choices can make big differences.

5.1 Brute-force

Probably the simplest to implement and the easiest solution for humans to grasp. The difference between the random and set boards in 9x9 shows the instability of this method. It relies heavily on some sort of luck. Although the algorithm used in this test is not fully brute-force but a combination of that and a stochastic optimization by randomizing the order numbers are added to their cell, the difference between best and worst time is significant. The worst case can be up to and over 20 times worse than the best case. This makes the algorithm unreliable in time measurements but in most cases still come up with a solution within a reasonable amount of time compared to the others tested.

Figure 5.1. Brute-force random boards results. Left 9x9. Right 16x16.

As the complexity of the problem grow so does the reliability of this method. On the 16x16 sudokus the difference between worst and best case grows even more. This can be seen in Appendix C section A.2, where the best case is 285 times faster than the worst.

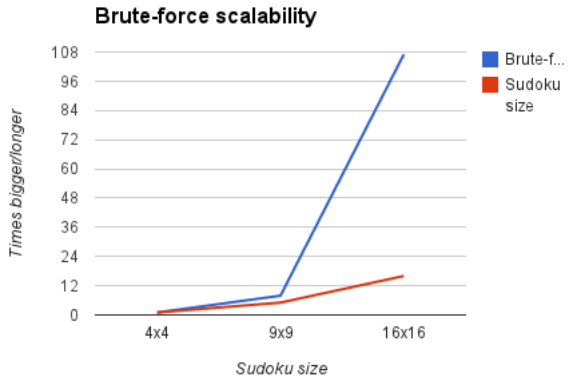


Figure 5.2. Scalability of Brute-force algorithm across different sizes.

Figure 5.2 shows a graph comparing the increase in sudoku size and the solving time for brute-force. Note that the 4x4 sudoku is held as a starting point and therefore both size and time are set to 1.

5.2 Simulated Annealing

Simulated annealing was by far the most unstable in testing. Results varied a lot which makes it tough to draw a qualified conclusion about the actual performance of the algorithm. It was the slowest in all tests, and had to be removed from the 16x16 sudoku test because it did not complete the puzzles within a reasonable amount of iterations. On 100.000 iterations the test was terminated and the algorithm had not produced a valid solution. The smaller sudokus, however, it was able to solve but in some cases the algorithm got stuck on the last two cells. This was most likely

due to the reheating process not working properly and could not get us out of the bigger local minimas which resulted in a much longer solving time.
The scalability is also a huge let down for simulated annealing as it has the highest increase in solving time on problem size increase of all the tested algorithms. At an increase of 11.2 times the 4x4 solving time at the 9x9 sudoku simulated annealing scalability is way above the other two algorithms.

5.3 Dancing Links

Dancing links is the most efficient algorithm of the three tested. It did not only get an excellent time result compared with the other algorithms but also a splendid scaling result. However, dancing links is hard to understand compared with both brute-force and simulated annealing. The implementation of the algorithm is not very hard but the idea behind it is very unique and therefore hard to grasp.
As exact cover is a np-complete problem it is bound to be hard to solve for computers and therefore optimizations are necessary. The idea of using double linked lists which has the ability to both cover and uncover a column/row in constant time by just moving its neighbors pointers and to store only zeros to speed up search time in the matrix grant obvious advantage in run time. Perhaps it is because of this dancing links is completely unrivaled by the others. The scalability were not far from being linear at 4x4 to 9x9 sudoku transfer and towards the bigger 9x9 to 16x16 change scalability still grew less than brute-force.

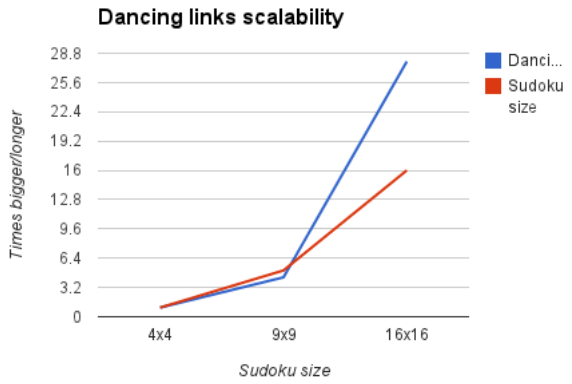


Figure 5.3. Scalability of Dancing links algorithm across different sizes.

The Dancing links curve goes below the sudoku size curve as seen in figure 5.3 because of the algorithms low solving time. As the sudoku increase the number of givens also increase and dancing links seem to be able to take advantage of this much better than other algorithms. However as the sudoku size increase even further the number of cells to solve increase enough to break this trend.

Chapter 6

Conclusion

It is easy to choose which algorithm that is the best, however all of them have their benefits and disadvantages. The brute-force algorithm is by far the easiest for a human to grasp and understand but it has worst cases that makes average time quite bad considering the fastest solving time. It also does not scale well when the puzzle gets larger because the worst case scales even more as the puzzle grows. It works perfectly fine for 9x9 sudoku but beyond that you should preferably consider alternatives if you want a fast solver.
Simulated Annealing was overall bad, with the worst results across all tests and not even completing the 16x16 test. However, this algorithm could potentially work better if the probabilistic function and iterations methods were rewritten. It tough to see how it scales because the 16x16 test failed. It is a very unreliable algorithm for solving sudoku in its current state.
Dancing Links is by far the best algorithm, it solves the puzzles the fastest and does not rely on guessing as much as the others, because of the constraints. With this comes more stable results, this also makes the data easier to interpret. It was not only the fastest across all test but also had the best scalability.

The purpose of the current study was to determine the best sudoku solver, in which we did. Dancing Links was by far the best algorithm for the tests that were chosen. A big part of this is because the structure of this algorithm is different from the two others. It does not rely on guessing and uses a intelligent data structure that fits the puzzle really well. Instead of relying on heavy iterations to get it right, it relies on constraint programming. From our observations Dancing links is a better sort of backtracker than brute-force, it guesses but gains a lot of benefit from the way backtracking works and its data structure.

Chapter 7

References

[1] Helmut Simonis. Sudoku as a Constraint Problem [webpage]. c2005. [retrived 11 April 2013].
Available at http://homes.ieu.edu.tr/bhnich/mod-proc.pdf
[2] Tim Kovaks. Introduction to sudoku solvers [webpage]. c2009.[retrived 11 April 2013].
Available at http://www.cs.bris.ac.uk/Publications/Papers/2000948.pdf
[3] George A. Miller. The Magical Number Seven, Plus or Minus Two: Some limits on our capacity for processing information [book]. c1956.
[4] Astraware Limited. Solving Sudoku: X-Wings [webpage]. c2008. [retrived 11 April 2013].
Available at http://www.sudokuoftheday.com/pages/techniques-8.php
[5] Astraware Limited. Solving Sudoku: Swordfish [webpage]. c2008. [retrived 11 April 2013].
Available at http://www.sudokuoftheday.com/pages/techniques-9.php
[6] Dimitris Bertsimas and John Tsitsiklis. Simulated Annealing [webpage]. c1993. [retrived 11 April 2013].
Available at http://www.mit.edu/~dbertsim/papers/Optimization/Simulated%20annealing.pdf
[7] Oracle. Nanotime java API [webpage]. c1996. [retrived 11 April 2013].
Available at http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html#nanoTime()
[8] Bertram Felgenhauer and Frazer Jarvis. Enumerating possible Sudoku grids [webpage]. c2005. [retrived 11 April 2013].
Available at http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf
[9] Jonathan Chu.A Sudoku Solver in Java implementing Knuth's Dancing Links Algorithm [webpage]. c2009. [retrived 11 April 2013].
Available at http://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/sudoku.paper.html
[10] Peter van Beek. Backtracking search algorithm [webpage]. c2006. [fetched 11 April 2013].

Available at https://cs.uwaterloo.ca/~vanbeek/Publications/survey06.pdf
[11] Wim Bohm. Dancing links, A backtrack data structure and algorithm by Donald Knuth [webpage, prestentation]. c2010. [retrived 11 April 2013].
Available at http://www.cs.colostate.edu/~cs420dl/slides/DLX.pdf
Released under Creative Commons Attribution licence.
[12] Donald E. Knuth. Dancing Links [webpage]. c2000. [retrived 11 April 2013].
Available at http://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/0011047.pdf
[13] Bret Wilson. Solving sudoku with Simulated Annealing [webpage]. c2011. [retrived 11 April 2013]. Available at http://www.cs.mercer.edu/courses/Laurie%20White/Old%20Things/CSC%20380/Homework%207%20Presentations/Solving%20Sudoku%20with%20Simulated%20Annealing.pptx
[14] Francesca Rossi, Peter van Beek,Toby Walsh. Constraint programming (CSP) backtracking, Handbook of Constraint Programming [webpage]. c2006. [retrived 11 April 2013].
Available at http://books.google.se/books?id=Kjap9ZWcKOoC&pg=PA21&redir_esc=y#v=onepage&q&f=false

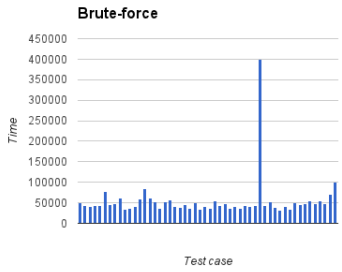
List of Figures

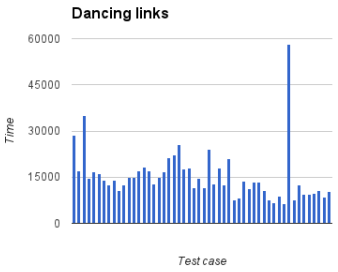
Appendix A

A.1 Random 4x4 data

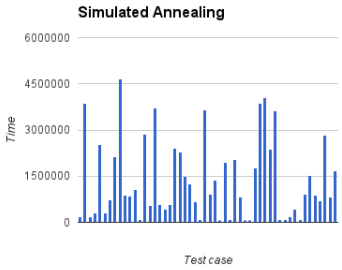
Data	Brute-force	Simulated Annealing	Dancing Links
Time Per Test (ns)	49178	187305	28652
	42764	3864115	17105
	40625	188588	35066
	42763	319444	14540
	44474	2527326	16677
	78258	302766	16250
	44901	716717	14112
	47468	2145875	12401
	62007	4653102	14112
	33783	883923	10691
	37204	867245	12402
	41480	1071227	14967
	59014	77402	14967
	85100	2875421	17106
	61579	562768	18389
	52599	3718719	17105
	36776	588854	12829
	52171	425070	14967
	56448	585005	16678
	40626	2405450	21382
	39770	2286995	22237
	646184	1481329	25658
	36349	1253827	17533
	50889	665828	17961
	33355	77402	11546
	41053	3661843	14540
	36777	910863	11546
	55165	1371427	23947
	44475	75691	12829

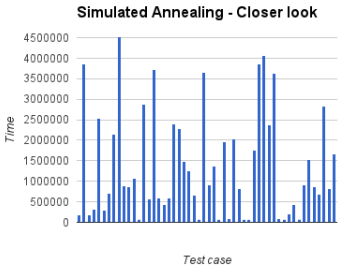
	47040	1960282	17961
	37632	80396	12401
	41480	2032124	20954
	37204	820205	7698
	43191	76119	8125
	41908	75692	13684
	44474	1756726	11119
	400694	3858555	13256
	44047	4056123	13257
	51744	2377226	10691
	38487	3627205	7698
	32927	82106	6842
	40625	77830	8980
	34639	195430	6414
	49606	427636	58158
	46184	79540	7698
	47468	920272	12402
	54738	1537350	9408
	47040	876653	9408
	53882	691059	9836
	47040	2835224	10691
	69705	829186	8553
	99640	1666067	10263
Avrage Time (ns)	48220	1339234	14582





A.1. RANDOM 4X4 DATA

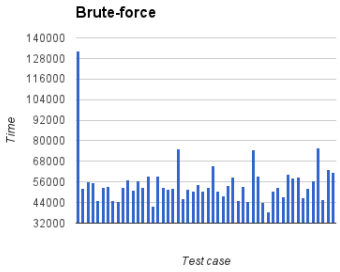


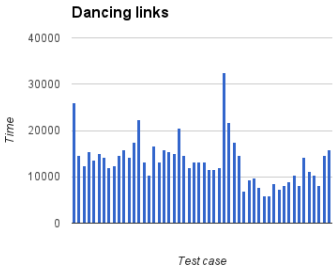


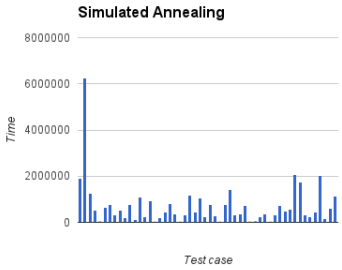
A.2 Set 4x4 data

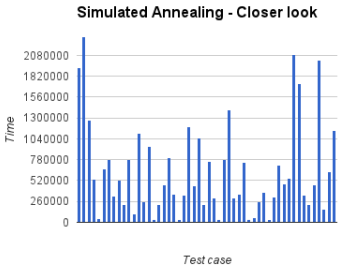
Data	Brute-force	Simulated Annealing	Dancing Links
Time Per Test (ns)	132139	1927781	26086
	52171	6266143	14540
	56021	1272215	12402
	55593	532834	15395
	44902	48750	13684
	52599	664118	14967
	53455	777014	14112
	45330	323720	11974
	44474	522571	12401
	52599	221942	14540
	57303	777014	15823
	51316	107765	14112
	56875	1109286	17533
	53027	257437	22237
	59441	945930	13257
	41908	37205	10263
	59442	216384	16677
	53027	463984	13256
	51744	806521	15823
	52172	347240	15394
	75264	36777	14967
	46185	342536	20526
	51744	1188827	14540
	50888	455005	11974
	54309	1050701	13257
	50461	224937	13257
	52599	757770	13256
	65428	300628	11546
	50461	37204	11546
	47896	781717	11974
	53882	1405210	32500
	59014	306187	21810
	45329	346385	17533
	53454	741947	14540
	44474	37632	6843
	74836	54737	9408
	59441	260857	9836
	44047	374181	7697
	38487	36349	5987
	50889	311318	5987

	52599	714151	8553
	47467	480662	7269
	60724	551222	8125
	58158	2087717	8980
	59014	1727647	10263
	47040	335694	8125
	52171	227075	14112
	56875	462702	11118
	75692	2018867	10264
	45757	157798	8125
	63290	626058	14539
	61579	1143069	15822
Avrage Time (ns)	54087	618138	13205



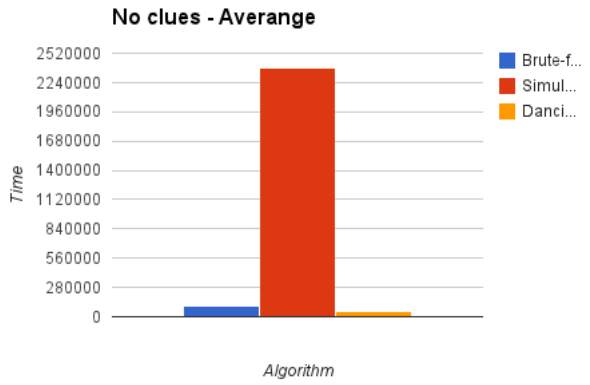



Simulated Annealing
8000000
6000000
4000000
2000000
0
Time
Test case


Simulated Annealing - Closer look
2080000
1820000
1560000
1300000
1040000
780000
520000
260000
0
Time
Test case

A.3 No clue 4x4 data

Data	Brute-force	Simulated Annealing	Dancing Links
Time Per Test (ns)	178324	5889824	64145
	102205	850995	46613
	103488	11675305	47040
	114606	3055456	47895
	100494	1173004	46612
	99639	920272	55593
	118883	1999623	47468
	120166	1477909	50461
	100494	2455056	46612
	102633	2036828	46185
Average Time (ns)	107871	2375996	48536

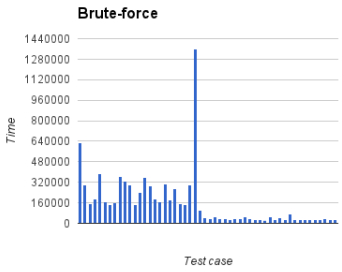


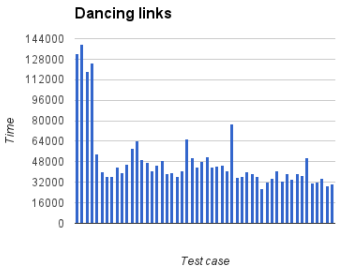
Appendix B

B.1 Random 9x9 data

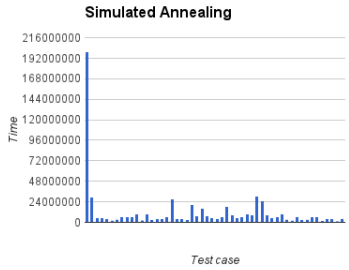
Data	Brute-force	Simulated Annealing	Dancing Links
Time Per Test (ns)	631190	199944408	132567
	297206	29858367	139837
	151810	5428406	118455
	190725	5664887	124870
	389576	4581687	53882
	165067	2330186	40197
	144541	3794410	36776
	160363	6522724	36777
	365201	6093806	43619
	327997	6164794	39770
	298917	9966474	45757
	148818	2637656	58159
	242897	9351106	64573
	361780	3306478	49605
	293785	4120268	47468
	193291	3994971	41053
	164640	6199860	45330
	304904	27877987	49178
	184311	4245565	38487
	266845	4130103	39771
	150100	3244471	36777
	149673	20924206	40626
	302766	7304014	65429
	1359025	16765023	50888
	99639	7180855	44046
	41480	5875712	47895
	35494	4640273	52172
	49179	6615522	43619
	38915	18676554	44474

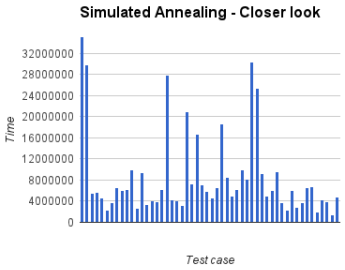
	38915	8543730	45329
	32072	5061922	41053
	34211	6303347	77830
	32928	9994270	35494
	53455	8231983	36777
	35922	30377089	40198
	30789	25408820	38488
	26941	9306632	36777
	24802	5005474	27369
	52599	6063444	32500
	29079	9513179	35066
	42336	3765759	40626
	25658	2263047	32928
	76547	6122030	38487
	31217	2764663	34211
	28652	3709310	38487
	31645	6604831	37632
	30362	6791708	50889
	31218	1925643	31645
	26086	4237440	32500
	35066	3983425	35066
	29934	1388961	29080
	30362	4869058	30363
Avrage Time (ns)	138742	8166863	47672





B.1. RANDOM 9X9 DATA

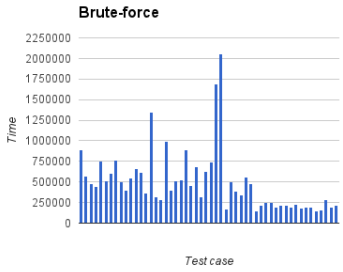


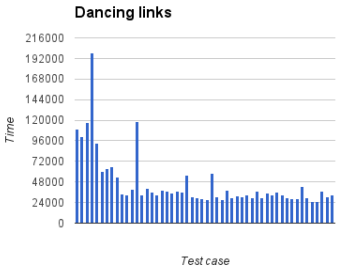


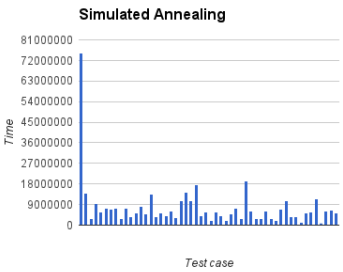
B.2 Set 9x9 data

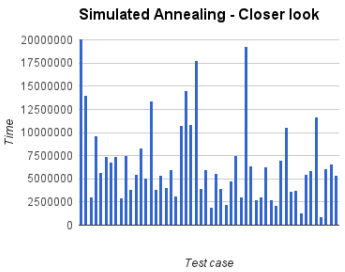
Data	Brute-force	Simulated Annealing	Dancing Links
Time Per Test (ns)	895041	75139410	109474
	568328	14015754	101349
	481090	3034501	117600
	442175	9658148	198850
	759481	5665743	92797
	509314	7422897	60297
	603821	6837036	63290
	769744	7455398	66283
	506320	2955817	53882
	398129	7477207	33783
	543953	3823916	33356
	667966	5449360	39343
	619217	8306392	118883
	360070	5104685	33356
	1352184	13406373	40626
	323293	3877799	35921
	281812	5391201	33356
	998101	4061255	38060
	404543	6040352	37204
	515729	3168352	35493
	527703	10755461	37204
	895896	14517370	36349
	452011	10844409	55593
	684217	17808454	30790
	317306	3983853	29079
	625631	5942851	28224
	741092	1906827	26941
	1685311	5552419	58586
	2059065	3972307	31218
	166350	2219856	27369
	505893	4794649	38059
	393853	7552043	29079
	341680	3082397	31645
	554643	19326560	31217
	480234	6359795	32928
	151383	2741143	29507
	213817	3055884	36776
	256582	6287953	29934
	251022	2720617	35493
	190725	2110809	32928

	218521	6997827	35921
	214245	10508715	33356
	196712	3704179	29507
	231351	3718718	28651
	179607	1316262	28224
	199706	5489985	42763
	196285	5902653	29080
	149245	11626982	25231
	162501	918133	24803
	288227	6066866	37631
	194574	6571047	30790
	214673	5370247	33356
Avrage Time (ns)	474641	6519226	43755



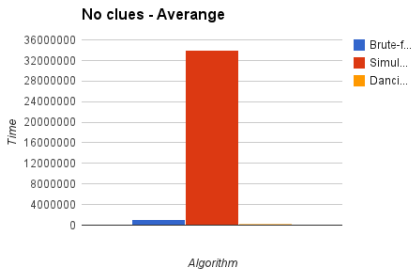


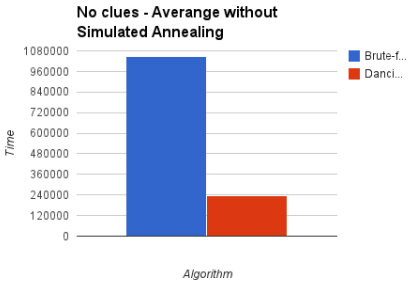




B.3 No clue 9x9 data

Data	Brute-force	Simulated Annealing	Dancing Links
Time Per Test (ns)	1188826	99497956	599973
	2987035	44510870	402405
	1117411	16635021	348095
	982279	15291390	401122
	1053266	14180821	144541
	1079780	86204051	144541
	757343	6646311	162073
	983134	28883786	133850
	758625	30833804	150528
	1226459	34946802	145823
Avrage Time (ns)	1048722	33935818	237391



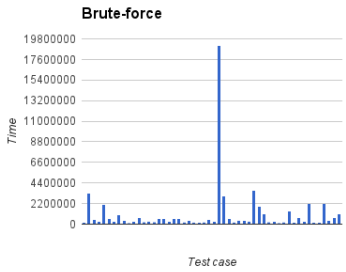


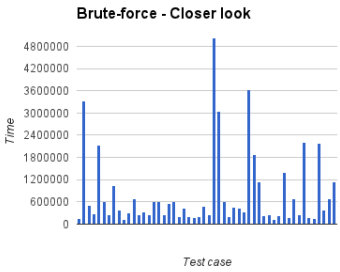
Appendix C

C.1 Random 16x16 data

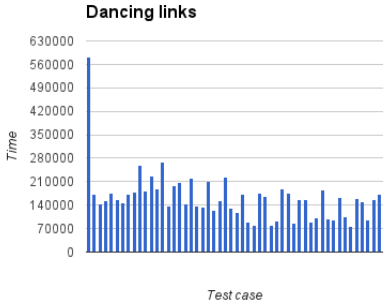
Data	Brute-force	Dancing Links
Time Per Test (ns)	152238	580729
	3322301	174047
	496485	142403
	289509	152238
	2138605	176186
	603393	156087
	261286	148390
	1039582	174047
	378457	178324
	126152	258719
	311319	183883
	683789	228358
	241186	187732
	325003	268982
	246745	138126
	611519	196712
	618789	208258
	257009	144541
	567472	219804
	599545	135988
	215101	135133
	437471	209541
	197567	124441
	168060	153521
	215101	222371
	482373	132567
	265562	118883
	19086656	174048
	3043482	90231

	612802	80823
	206548	174903
	447307	166350
	432767	80395
	325858	92369
	3620362	189870
	1872616	175759
	1134090	85955
	236910	156087
	254016	158226
	129146	89376
	216384	102632
	1397085	186876
	186449	99212
	682078	97073
	255298	162501
	2198474	105198
	174903	76119
	156514	159508
	2175382	149245
	374609	94935
	687210	157370
	1151622	171482
Avrage Time (ns)	741947	155394





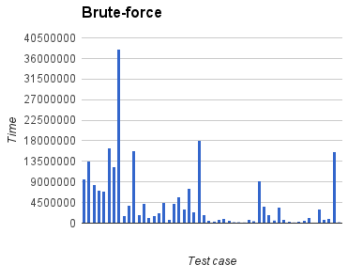
C.1. RANDOM 16X16 DATA

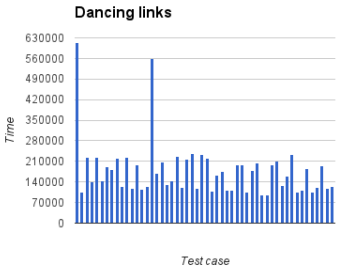


C.2 Set 16x16 data

Data	Brute-force	Dancing Links
Time Per Test (ns)	9636339	612801
	13579138105198	
	8338893	222798
	7295462	140264
	7013651	225364
	16494330	143685
	12297087	192863
	37959067	181317
	1587383	221943
	3805956	125297
	15863995	222371
	1877748	118027
	4397376	197140
	1184550	115889
	1646397	124014
	2169395	558492
	4557739	168060
	844580	208686
	4337507	131711
	5850909	144968
	3114898	225791
	7593951	121449
	2389628	218521
	18124904	235627
	1899129	118027
	549940	232633
	419511	220660
	726980	109047
	938232	163357
	539676	174475
	242042	112041
	237766	112896
	169343	196712
	719710	198423
	513591	104771
	9336139	179179
	3772173	205265
	1772122	96218
	703033	96218
	3499769	198851

	791126	211680
	326714	127435
	168916	159080
	379740	233917
	563196	106909
	1229452	110757
	132994	183884
	3180326	105626
	880074	122731
	1069516	194574
	15531295	118455
	233061	124870
Avrage Time (ns)	4087887	171438





C.3 No clue 16x16 data

Data	Brute-force	Dancing Links
Time Per Test (ns)	30202186	3585297
	15494090	2145447
	63501736	1585245
	3710594	1646396
	11108261	2704367
	14047399	1564718
	43457176	1627153
	10445853	1904261
	458853	1241426
	12850874	1211491
Avrage Time (ns)	17664554	1802376