Scaling of popular Sudoku solving algorithms

DD143X: Degree Project in Computer Science

Authors

Markus Videll <mvidell@kth.se>
Mattias Palo <mpalo@kth.se>

Supervisor

Alexander Kozlov <akozlov@csc.kth.se>

Examiner

Karl Meinke <karlm@csc.kth.se>
Örjan Ekeberg <orjan@csc.kth.se>

University School

KTH CSC

2014-04-29
Abstract

In this bachelor thesis we study $6 \times n^2$ popular Sudoku solving algorithms, found through Google, to find the algorithm that has the slowest growth. The algorithms we tested are: brute-force, a pen-and-paper method, two exact cover reductions in Python and Haskell, a SAT reduction, and a constraint satisfaction algorithm. The algorithms were tried by solving Sudoku puzzles of sizes $n = \{2, 3, 4, 5\}$ where we measured the solution time. We conclude that the slowest growing algorithm, by far, is the SAT reduction, followed by the exact cover reductions. Brute-force grows, unsurprisingly, the fastest.
# Contents

1 **Statement of collaboration** .................................................. 4

2 **Introduction** ........................................................................... 4

3 **Background** ............................................................................ 4  
   3.1 Terminology ........................................................................... 4  
   3.2 Sudoku .................................................................................. 5  
   3.3 Algorithms ............................................................................ 6  
      3.3.1 Brute-force method ........................................................... 6  
      3.3.2 A Pen-and-Paper Algorithm ............................................. 6  
      3.3.3 Exact Cover Reduction ...................................................... 7  
      3.3.4 SAT Reduction ................................................................. 7  
      3.3.5 Constraints Satisfaction Problem .................................... 8

4 **Method** .................................................................................. 8  
   4.1 Selecting algorithms ............................................................... 8  
   4.2 Generating Sudoku puzzles ................................................... 9  
   4.3 Preparing the algorithms ....................................................... 9  
   4.4 Running the algorithms ........................................................ 9  
   4.5 Plotting the data .................................................................... 10

5 **Results** .................................................................................. 10  
   5.1 Graphs .................................................................................. 10  
   5.2 Interpretation ......................................................................... 13

6 **Discussion** ............................................................................ 13  
   6.1 Analysis ................................................................................ 13  
   6.2 Possible improvements .......................................................... 14  
   6.3 Conclusions .......................................................................... 14

**Appendices** .............................................................................. 17  

**Appendix A** Time distributions .................................................. 17  

**Appendix B** Hardware specifications .......................................... 23  

**Appendix C** Code ..................................................................... 24  
   C.1 Brute-force .......................................................................... 24
1 Statement of collaboration

- Markus has written most of this report and ran the algorithms.
- Mattias wrote most of the converters, prepared the algorithms and created the plots.
- Remaining work have been done together.

2 Introduction

Sudoku puzzles are a popular time sink, similar to crosswords. People solve them while riding the train, or are waiting for their dentist appointment. There is a much more serious side to it. There are several competitions in Sudoku solving including The World Sudoku Championship[1]. There are even claims that solving Sudoku puzzles have a positive effect on brain power[2], though there are conflicting reports[3].

In this report we have chosen to study 6 popular \( n^2 \times n^2 \) Sudoku solving algorithms and we are going to examine how fast each algorithm scales as the Sudoku puzzle grows. These values will then be plotted in a graph and the exponential function approximated. The algorithms will then be compared to each other to see which one exhibits the slowest growth. This report does not study properties other than solution time, e.g. memory usage.

3 Background

3.1 Terminology

Box:
A region that no symbols can be in at the same time. It is a 3 \( \times \) 3 square in the standard 9 \( \times \) 9 puzzle.

Cell:
One of the individual squares that can be a symbol, most often digits between 1 – 9.

Clue/Given:
One of the given digits that you are not allowed to change. They are the clues for you to use when solving the puzzle.

Symbol:
The symbol that is put into a cell. In the standard 9 \( \times \) 9 puzzle the symbols are
commonly the digits 1 – 9. In bigger puzzles letters may be used as well as digits. There are \( n^2 \) symbols in a \( n^2 \times n^2 \) Sudoku puzzle.

**Ambiguous puzzle:**
A puzzle with more than one solution.

### 3.2 Sudoku

![Sudoku board](image)

Figure 1: A standard 9 \( \times \) 9 Sudoku board. To the left is the original board and to the right is its solution in red[4].

Sudoku is a game with simple rules. The standard game is a 9 \( \times \) 9 board with 3 \( \times \) 3 boxes. The board is initially filled with a number of clues to get you started, and your goal is to fill all cells with the digits 1 – 9 so that each number only occurs once in each row, column, and box, until there are no empty cells left. See figure 1 for an example puzzle.

As stated above, the most common boards are 9 \( \times \) 9, but it is actually possible to create boards with size \( n \times n \), where \( n \) is the width in cells. It is important to note that if \( n \) is not a squared number, the smaller boxes will not be square. That is not a big issue when solving by hand, but when we are searching for Sudoku solving algorithms we have limited our searching and testing to puzzles with dimension \( n^2 \times n^2 \).
Figure 2: A $6 \times 6$ Sudoku puzzle[5], commonly referred to as “Kids Sudoku”. Notice that the boxes are not square when $n$ is not a square number.

3.3 Algorithms

In this section we describe the basic idea behind selected algorithms shown in Table 1. Main principles are broadly explained and only some implementation details are outlined here.

<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm</th>
<th>Language</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brute-force</td>
<td>Python</td>
<td>C.1</td>
</tr>
<tr>
<td>3</td>
<td>Exact Cover Reduction</td>
<td>Haskell</td>
<td>[7]</td>
</tr>
<tr>
<td>4</td>
<td>Exact Cover Reduction</td>
<td>Python</td>
<td>[8]</td>
</tr>
<tr>
<td>5</td>
<td>SAT Reduction + MiniSat</td>
<td>C</td>
<td>[9, 10]</td>
</tr>
<tr>
<td>6</td>
<td>Constraint Satisfaction Problem</td>
<td>Java</td>
<td>[11]</td>
</tr>
</tbody>
</table>

Table 1: The selected algorithms

3.3.1 Brute-force method

To solve a Sudoku puzzle using a brute-force method, one could do that by looping through each cell, one by one, and picking a number between 1 to 9. This could be done either in a specified sequential or a random order. If an invalid state is reached, i.e. there is a collision of symbols, it will backtrack and make a new choice. By doing this exhaustive search you are guaranteed a solution to the puzzle, provided there is one[12].

3.3.2 A Pen-and-Paper Algorithm

A simple algorithm that is often used when solving by hand. Each cell has a list of potential candidates that starts with all possible symbols. By checking the row, column, and box, the conflicting values are removed from this list. When there is only one candidate left, that cell is set[6].

6
This however is not enough to guarantee a solution, and the algorithm may come to a halt even when there are unfilled cells. When this happen the algorithm will choose the cell with the fewest possible candidates and pick one candidate at random. It will then continue doing logical deductions until it either finds a correct solution or an invalid state, in which it will backtrack[6].

3.3.3 Exact Cover Reduction

Exact Cover is one of the first known NP-complete problems and was proven NP-complete by Richard Karp in 1972. You are given a family of sets, \( C \), and the problem is to find a subset, \( C' \subseteq C \), so that each element in the sets in \( C \) appears exactly in one member of the sets in \( C' \)[13]. Sudoku can be reduced to Exact Cover and, since \( n^2 \times n^2 \) Sudoku itself is NP-complete[14], it can be reduced in polynomial time.

The reduction is done by creating a big matrix \( C \), where each row is \( C' \). Since this is a big matrix it is easier to represent the elements with indicator variables. A 1 means that the element is present in this subset, and a 0 means that it is not[15].

For a standard 9 \( \times \) 9 Sudoku board, the matrix will have 4 \( \cdot \) 81 columns. The first 81 columns is a unique identifier for each cell. For the Exact Cover to be solved, exactly one row for a given cell will be chosen. The next 81 columns represents the row index of that cell. The last 162 columns is the similarly representing the columns and boxes of the Sudoku puzzle[15]. For example, a one in column 82 means that the cell in the first row is set to 1. A one in column 83 means that the cell in first row is set to 2, etc.

The clues in the Sudoku puzzle will each be represented by a single row in the matrix. Since each cell has a unique ID and there is only be one row of them, these rows are forced to be a part of a solution.

But each empty cell will have to be represented by \( n^2 \) rows, one for each symbol. The dimensions of the final matrix is given by

\[
\text{(number of clues} + 9 \cdot \text{number of empty cells}) \times 4 \cdot 81
\]

for a standard 9 \( \times \) 9 Sudoku board.

The Exact Cover problem is then solved by Donald Knuth’s Algorithm X with Dancing links[15].

3.3.4 SAT Reduction

The Boolean Satisfiability Problem (SAT) is another popular NP-complete problem, and it is one of the most researched problems. Because of this there are several SAT solvers out there, and there are even competitions to see who can create the fastest[16].
The reduction works by dividing the problem into the following rules:

1. Each row must contain each symbol once
2. Each column must contain each symbol once
3. Each box must contain each symbol once
4. Each cell in the puzzle can contain only one number

The program then formulate clauses, where each variable is a symbol in the puzzle.

\((a \lor b \lor \ldots \lor x) \land (\neg a \lor \neg b) \land \ldots \land (\neg a \lor \neg x) \land (\neg b \lor \neg c) \land \ldots \land (\neg b \lor \neg x) \land \ldots \land (\neg w \lor \neg x)\)

This formula means that only one of the symbols \(a\) to \(x\) can be true. The formula is used for each cell, row, column, and box. The given clues will be encoded by appending clauses consisting of a single variable corresponding to that number in that cell[9].

The reduction we used is only a reduction and does not solve the formula that is outputted. We used MiniSat which is currently one of the best solvers[10].

3.3.5 Constraints Satisfaction Problem

A Constraints Satisfaction Problem (CSP) is a problem that can be described as a set of variables, domains for the variables, and constraints. Sudoku can easily be represented as a CSP. Each cell is a variable, and the domain is all the used symbols, 1 – 9 in a standard Sudoku puzzle. The constraint that each cell can not be the same as any other cell in each row, column, or box[11].

The program then solves the puzzle by making a choice, and then backtracks if it encounters an invalid situation. This works better than a simple brute-force as it also does a forward checking to constrain the variables domains[11].

This implementation also provides two additional heuristics for harder problems. The Minimum-Remaining-Values that selects which variable to be assigned next, and Least-Constraining-Values which selects what value that should be assigned[11].

4 Method

4.1 Selecting algorithms

For the comparison study, 6 different implementations where selected among \(n^2 \times n^2\) Sudoku solvers found on Google. Google was used because we wanted to find popular
algorithms. Since scaling is independent of the language used in the implementation we did not hesitate to choose algorithms in different languages. For a list over selected algorithms, see Table 1.

4.2 Generating Sudoku puzzles

To test the algorithms we needed unambiguous puzzles in different sizes. We could not generate our own puzzles because of time constraints so we had to find already generated puzzles.

All Sudokus we used are generated by the same program and are presented on the same website[17]. This program is able to generate Sudokus of different sizes and difficulties, though we noted that the bigger the generated Sudoku is, the easier and more clues they have.

We downloaded puzzles with the sizes $n = \{2, 3, 4, 5\}$ and difficulty “Easy”, and got about 150 – 200 puzzles for each size.

The “Easy” difficulty was chosen to guarantee uniform conditions for tests in cases for all sizes. Because of time constraints we also needed the algorithm to finish in a reasonable time.

4.3 Preparing the algorithms

Before testing the algorithms we made some small changes to them. Since they represented the Sudoku puzzles differently, we created a converter for almost all of them. On some algorithms we also made changes so that you could input a file with puzzles and it would solve all of them, and others would be run together with a bash script that looped over all puzzles. The solution would be printed to a separate file together with the solution time.

4.4 Running the algorithms

We used one of our own computers to run the algorithms, and it would stay on during the night. To speed up the testing we ran several algorithms at the same time. Since no algorithm makes use of multiple cores this was deemed to not interfere with the scaling of each algorithm so long as the CPU load stayed below 100%.

The computer we used is an x86_64 machine with Arch Linux as operating system. It has 4 physical CPU cores but uses Hyper-threading to get two threads on each core. Each core is clocked at about 3.7GHz and it has 16GB of RAM. For more hardware information, see Appendix B.
<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm</th>
<th>Compiler/Interpreter</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brute-force</td>
<td>PyPy</td>
<td>2.2.1</td>
</tr>
<tr>
<td>2</td>
<td>Pen-and-Paper</td>
<td>GNU Octave</td>
<td>3.8.0</td>
</tr>
<tr>
<td>3</td>
<td>Exact Cover Reduction</td>
<td>GHC</td>
<td>7.6.3</td>
</tr>
<tr>
<td>4</td>
<td>Exact Cover Reduction</td>
<td>PyPy</td>
<td>2.2.1</td>
</tr>
<tr>
<td>5</td>
<td>SAT Reduction + MiniSat</td>
<td>GCC</td>
<td>4.8.2</td>
</tr>
<tr>
<td>6</td>
<td>Constraint Satisfaction Problem</td>
<td>OpenJDK</td>
<td>1.7.0.51</td>
</tr>
</tbody>
</table>

Table 2: A list of the different compilers and interpreters we used to test the algorithms.

4.5 Plotting the data

The data was gathered and plotted using Matlab. The plotted values are the median value of all puzzles in that size, which were chosen to eliminate extreme cases. We used Matlab to approximate the growth rate as an exponential function, \( f(n) := \alpha \exp(\beta n) \), between each data point. The exponential function was chosen because \( n^2 \times n^2 \) Sudoku is NP-complete and there exists no known algorithm to solve it in polynomial time.

5 Results

5.1 Graphs

Median values of solving times for selected algorithms are shown in the plots below. Exponential fit \( f(n) \) for the test data is presented in every graph for each algorithm. Lastly, we present two combined graphs for easier comparison. The first one depicts the scaling of each algorithm, \( \beta n \), and the second one is a combined graph of \( f(n) \). Distributions of solving times are shown in Appendix A.
Algorithm 1: Brute-force

\[ f(n) = 2.342298 \times 10^{-21} \times e^{14.302321n} \]

Algorithm 2: Pen-and-paper

\[ f(n) = 9.109046 \times 10^{-10} \times e^{6.749746n} \]

Algorithm 3: Exact Cover Reduction (Haskell)

\[ f(n) = 4.735181 \times 10^{-11} \times e^{5.776683n} \]

Algorithm 4: Exact Cover Reduction (Python)

\[ f(n) = 5.810796 \times 10^{-14} \times e^{6.540075n} \]
Algorithm 5: SAT Reduction

\[ f(n) = 7.931658 \times 10^{-5} \exp(1.665457n) \]

Algorithm 6: Constraint Satisfaction Problem

\[ f(n) = 6.371785 \times 10^{-12} \exp(8.069722n) \]

Scaling (all algorithms, beta variable)
5.2 Interpretation

The SAT reduction is the slowest growing algorithm, and the second slowest growing is the Exact Cover reductions. The Python version grows a bit faster than the Haskell version, as we can see on the approximated exponent in the function, but they are mostly the same. The Python version actually solves smaller puzzles faster than Haskell.

The other algorithms, Pen-and-paper, CSP, and brute-force methods are all incredibly slow as the Sudoku puzzle grows. The Pen-and-paper and CSP were able to solve all problems with size $n = 4$ after a couple of days, but the brute-force had only managed to solve three puzzles after the same amount of time.

6 Discussion

6.1 Analysis

In this paper, we studied the scaling of $6 \times n^2 \times n^2$ Sudoku solving algorithms. The results clearly show that the SAT reduction has the slowest growth rate of the studied algorithms, many times slower than Exact Cover in Python. We are not surprised that the SAT reduction was fast, it is as we mentioned a very studied problem, but we are
surprised that it was so much faster than the other algorithms. The SAT reduction is also much less popular than the other algorithms like Exact Cover reductions and Pen-and-paper methods.

The Pen-and-paper method and CSP algorithms were among the slowest in solution time, but that may be attributed to the language used. In scaling however they were not far away from the Exact Cover reductions.

It is also important to note that just because the SAT reduction grows slowest, it does not mean that other algorithms are useless. There may be other scenarios where other algorithms are more suited, e.g. memory usage, puzzle generation, ease of implementation, and difficulty classification. For example when \( n \leq 3 \) the solution time is below a second for almost all algorithms, and you should probably focus on other properties instead of scaling or solution time. Other properties were not studied at all in this report however.

### 6.2 Possible improvements

There are a few things that could have been done better. We used the difficulty classification from the website that generated the Sudoku puzzles, but we do not know exactly how they are classified as such. We also would have liked to use a harder classification than “Easy”, but they were more difficult to find.

Similarly, we should also have tried with more data points. While four or five data points is enough to see a drastic difference in solution time, it is not necessarily enough to accurately approximate the coefficients of the exponential function. To deal with this the algorithms could have been implemented in a faster language like C instead, and they may have been able to solve bigger puzzles. Because of time constraints we were not able to do this.

The last problem is the implementations. They were found through Google and are created by ordinary people and are not necessarily optimal, except for MiniSat. We can see some difference for example between the Python and Haskell implementations. Python scales a bit faster than the Haskell implementation, even though they use the same method. This difference however is not that notable, and does not interfere with our results.

### 6.3 Conclusions

We can draw the conclusions that the SAT reduction scales the slowest, followed by Exact Cover reductions. The brute-force method grows incredibly fast and should not be used for puzzles with large \( n \) values.
References


Appendix A  Time distributions

In this section we show the time distribution to solve for all algorithms, sizes, and puzzles. The red line shows the median value. Do note the exponent on some of the graphs.

Algorithm 1: Brute-force
Algorithm 2: Pen-and-paper
Algorithm 3: Exact Cover Reduction (Haskell)
Algorithm 4: Exact Cover Reduction (Python)
Algorithm 5: SAT Reduction
Algorithm 6: Constraint Satisfaction Problem
Appendix B  Hardware specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>x86_64</td>
</tr>
<tr>
<td>CPU op-mode(s)</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Byte Order</td>
<td>Little Endian</td>
</tr>
<tr>
<td>CPU(s)</td>
<td>8</td>
</tr>
<tr>
<td>On-line CPU(s) list</td>
<td>0-7</td>
</tr>
<tr>
<td>Thread(s) per core</td>
<td>2</td>
</tr>
<tr>
<td>Core(s) per socket</td>
<td>4</td>
</tr>
<tr>
<td>Socket(s)</td>
<td>1</td>
</tr>
<tr>
<td>NUMA node(s)</td>
<td>1</td>
</tr>
<tr>
<td>Vendor ID</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>CPU family</td>
<td>6</td>
</tr>
<tr>
<td>Model</td>
<td>60</td>
</tr>
<tr>
<td>Model name</td>
<td>Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz</td>
</tr>
<tr>
<td>Stepping</td>
<td>3</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>3760.984</td>
</tr>
<tr>
<td>CPU max MHz</td>
<td>3900.0000</td>
</tr>
<tr>
<td>CPU min MHz</td>
<td>800.0000</td>
</tr>
<tr>
<td>BogoMIPS</td>
<td>6800.82</td>
</tr>
<tr>
<td>Virtualization</td>
<td>VT-x</td>
</tr>
<tr>
<td>L1d cache</td>
<td>32K</td>
</tr>
<tr>
<td>L1i cache</td>
<td>32K</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256K</td>
</tr>
<tr>
<td>L3 cache</td>
<td>8192K</td>
</tr>
<tr>
<td>NUMA node0 CPU(s)</td>
<td>0-7</td>
</tr>
</tbody>
</table>

Table 3: The computer specifications of the computer (from lscpu)
Appendix C  Code

C.1  Brute-force

from __future__ import division

import sys
import string
import time

symbols = '123456789' + string.ascii_uppercase + string.ascii_lowercase

def isValid(spuzzle, cellIndex, symIndex):
    if symIndex >= limit:
        return False
    row = cellIndex // limit
    col = cellIndex % limit
    box = row//boxsize*boxsize*limit + col//boxsize*boxsize

    for i in range(limit):
        # Check row
        if spuzzle[row*limit + i] == symbols[symIndex]:
            return False
        # Check column
        if spuzzle[col + i*limit] == symbols[symIndex]:
            return False
        # Check box
        if spuzzle[
            box + (i//boxsize*limit + i%boxsize)
        ] == symbols[symIndex]:
            return False
    return True

def solve(puzzle):
    """Solve by setting a digit to a value and moving on until either it is
    solved or it reaches an invalid state. At that point it will backtrack.""

    # The current cell index and symbolindex
    cellIndex = 0
    # Skip to next cell that is not a clue
    while cellIndex < len(puzzle)-1 and puzzle[cellIndex] != '.':
        cellIndex += 1
    symIndex = 0
# The puzzle we do changes in. Since we do not want to change
# the given clues we need to keep a backup. Since strings are immutable
# we convert it to a char array.
spuzzle = list(puzzle)
del spuzzle[-1] # Remove the trailing newline

global limit, boxsize

# The number of symbols is the square root of the length. It is
# also guaranteed to be an integer by the generator we used.
limit = int(len(spuzzle) ** (1/2))

# The boxsize is the square root of row length
boxsize = int(limit ** (1/2))

while cellIndex < len(puzzle)-1: # -1 for newline
    if isValid(spuzzle, cellIndex, symIndex):
        # Placing symbol symIndex is valid, proceed to next cell.
        spuzzle[cellIndex] = symbols[symIndex]
        cellIndex += 1
        # Skip to next cell that is not a clue
        while cellIndex < len(puzzle)-1 and puzzle[cellIndex] != '.':
            cellIndex += 1
    symIndex = 0
else:
    symIndex += 1
    if symIndex >= limit:
        # The symIndex has exceeded the number of symbols used in game
        # => the current solution is invalid. Backtrack.
        spuzzle[cellIndex] = '.
        cellIndex -= 1
        # Skip to previous cell that is a clue
        while puzzle[cellIndex] != ',':
            cellIndex -= 1
        # symIndex is the currently placed symbol +1 (since it is invalid)
        try:
            symIndex = symbols.index(spuzzle[cellIndex])+1
        except:
            for i in range(limit):
                print(spuzzle[i*limit:(i+1)*limit])
        print(cellIndex, symIndex, symbols)

25
raise

return ''.join(spuzzle)

def main():
    """Solves Sudoku puzzles by brute-force"""
    if len(sys.argv) < 2:
        print("Usage: python3 brute-force.py file file file...")
        return

    files = sys.argv[1:]  
    for fname in files:  
        with open(fname) as f, open(fname+'.solved1', 'w') as s:
            for line in f:
                t = time.clock()
                solved = solve(line)
                s.write("%s %f\n" % (solved, time.clock()-t))  
                s.flush()
                print("%s %f" % (solved, time.clock()-t))

if __name__ == '__main__':
    main()