Sudoku Solver

Adam Ekström, Felix Magnell

Fall 2018

1 Introduction

In this project we have studied different approaches on solving the Sudoku puzzle and created two different algorithms doing so.

The Sudoku puzzle is a game where numbers are placed in a grid. The game board consists of a 9x9 grid with at least 17 initial numbers placed. The 9x9 grid is divided into smaller "sub-grids" that consists of 3x3 grids.

The natural constraints of Sudoku is that the numbers 1 to 9 should be placed in a way that every row, column and sub-grid consist of every number exactly once. [4]

2 Initial study

We have searched for different heuristics that is suitable for solving the Sudoku puzzle. We decided to implement two types of heuristics, Backtracking algorithm and Rule-Based algorithm. The Backtracking algorithm is quite straight forward in what to be done. For the Rule-Based algorithm we decided that every cell will be assigned a list of possible candidates, a *candidate-list* which is a list of numbers that at the current state of the board wont conflict with any of the games constraints. We also had to decide on which rules we wanted to implement to get a functioning and efficient solver.

We used a Sudoku dedicated web page as our resource, a website called Hodoku, which covers most Sudoku techniques used by humans to solve the Sudoku puzzle. We went through the different techniques and picked the ones we found necessary to implement as rules for our solver.

2.1 Rule-Based Algorithm

We decided to use the following three rules, each based on a solving technique:

- Naked Single
- Hidden Single

- Naked Pair

The *Naked Single* rule means that for a specific cell, there is only one candidate in its candidate-list. Thus the candidate should be placed in that specific cell. Whenever a cell is assigned a digit, every other cell in the same sub-grid, row and column will have that digit removed from its candidate-list. We decided to implement it since it is a necessity for solving a Sudoku puzzle with our candidate-list based approach. It was also easy to implement and powerful in the sense that it fills a square without having to check a lot of things, this rule can solve easy Sudoku-puzzles on it's own very fast. [2]

The *Hidden Single* rule means that when a cells candidate is the only one of its kind in the same sub-grid, row or column, that candidate can be assigned to its cell. Since no other cells in the same sub-grid, row or column contains the same digit as a candidate it wont inflict any constraints. [2]

This rule combined with Naked Single should be able to solve most Sudoku-puzzles found in newspapers. [3]

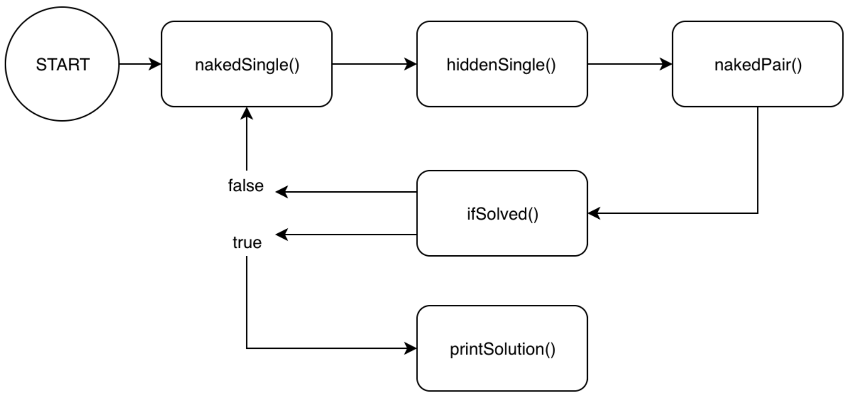
Finally the third rule is the *Naked Pairs*. This rules states that for a pair of cells in a sub-grid, row or in a column that contains only the same two candidates in its candidate-list, we can remove those two digits from every other cells, in the same sub-grid, row and column, candidate-list. [1]

This rule combined with the other two allows for solving even harder puzzles.

3 Design

Before we started to code the actual solver, we made flow charts on how we wanted the program to function.

3.1 Rule-Based Design



The algorithm iterates through the functions that applies the rules with the same name. The method *ifSolved()* checks if the puzzle is solved or not. When solved the solution to the puzzle is printed.

3.2 Backtrack Design

The design for this algorithm was quite straight forward. What we had to plan was what our base case would be, so that the recursive calls would end. We knew that we had enough experience in programming to implement this algorithm without any deeper planning.

4 Implementation

We used C++ to implement our algorithms and QT to create a GUI application. [7]

4.1 Backtracking

The backtracking algorithm uses a naive approach that starts in the cell 0x0 and checks if that cell contains a digit, if not it tests all possible numbers 1 through 9 until one accepted (not necessarily correct) digit is found.

solveSudoku() is a recursive function that iterates through the matrix, for every cell it checks whether a number is legal, by checking the house if the number already exist there. It starts trying from 1 and goes through 9. When a legal number is found it recursevly tries the next until a solution is found. If there are no legal numbers to place it goes back (therefore "backtracking") and tries the previous. To check if a digit is legal we use a help-function *checkLegal()* that takes coordinates to the cell we are checking and the number we are trying out as arguments.

4.2 Rule-Based Implementation

First we started with a basic 9x9 matrix printed to the console. We went through each function that we did flowcharts of. We started to implement *findCandidated().*

findCandidated() goes through the board and for every cell puts it's candidates in a list called candidate list. For a specific cell it checks every element in the row, column and sub-grid. if a number is not found in any of these checks it puts that number in the candidate list for the specific cell. This is a very important function as every rule is based on checking candidates.

Then we implemented the rules, ***nakedSingle()***. The idea is simple, go through the candidate list and check if there are any singles in the list, if there are assign that digit to its cell. This means if a cell only has one candidate and thus only one possible number that it can have, it must be the one to be placed on the board and remove all the candidates from the candidatelist for that cell.

hiddenSingle() contains three sub-functions that works similar, one for each member of a "house": a row, column and a sub-grid. When checking a row we call *hiddenSingleRow()* that checks a row for singles by going through each cell in the row checking its candidates and adding every occurrence of a each number in a list called *trackSingles*. When we have gone through the row we check the list if there are any "ones" in the list. If we find one cell that had a one this means that a that cell was alone in having this candidate and therefore should be placed in the board. The same strategy was used for *hiddenSingleCol()* and *hiddenSingleBox()*, only it checks collumns and sub-grids instead of rows.

nakedPair() consists of three sub-functions just as *hiddenSingle()*, that is: *nakedPairBox()*, *nakedPairRow()* and *nakedPairCol()*. For every house, we check if there are two cells that consists of the same two elements and doesn't have any more candidates than those, (a pair) we can remove those element as candidates from every other cell in the house. So if we take *nakedPairBox()* as an example, we go through every cell in a sub-grid and checks if the number of candidates is exactly 2, then we flag this index as a "double" meaning it's a potential pair whit another cell. Then we go through all the flagged indexes and checks if it's the same number on both, if it is we remove these numbers from all the cells candidates list in the house. Another fundamental function we implemented was *updateCandidates()* that takes row, column and number as argument and removes that number from the house. This function is called whenever a cell is assigned a digit or a candidate is removed from a cells candidate list.

numToCordConverter() is used to convert between a cell number (we named every cell from 1 to 81 for convenience) to easily be able to refer to them by a number instead of coordinates x and y. We also had a function ***cordToNum()*** to be able to go between coordinates and index-number back and forth.

When we implemented each function we generated a matrix that we knew should be affected from the first run, for example when we implemented Naked Single we created a matrix that had a specific cell where only one digit was possible to enter.

5 Testing

We created multiple matrices that were designed for each function. When we tested a specific function we made sure that the matrix changed to see if the algorithm behaved correctly. We also did a print function for our candidate-list to check if it was updated correctly.

When we tested *nakedPair()* the a sub-grid in the initial matrix looked like the left sub-grid in *figure 1*, and after the method had applied its rules the sub-grid was checked if it would look like the right sub-grid in *figure 1*.

When a function was complete we tested it with various Sudoku-puzzles, we tried all the different difficulty levels from Svenska Dagbladet SvD. [5]

For the ultimate test we used our algorithms on a Sudoku-puzzle had a Sudoku-puzzle, made by Arto Inkala, a Finnish mathematician that states that this is the hardest Sudoku-puzzle that has ever been created. [6] Our back-tracking algorithm did manage to solve it, but our rule-based algorithm did not, we would need more rules to manage the edge cases generated by this Sudoku-puzzle.

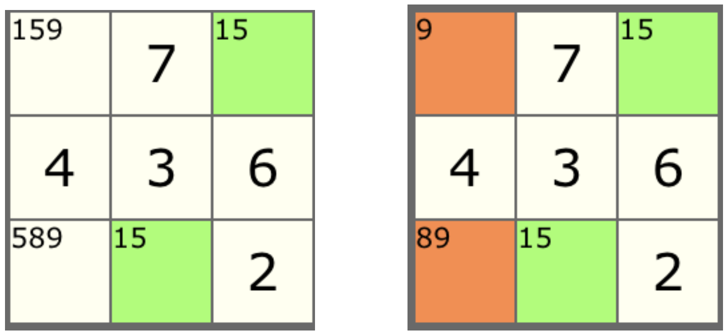


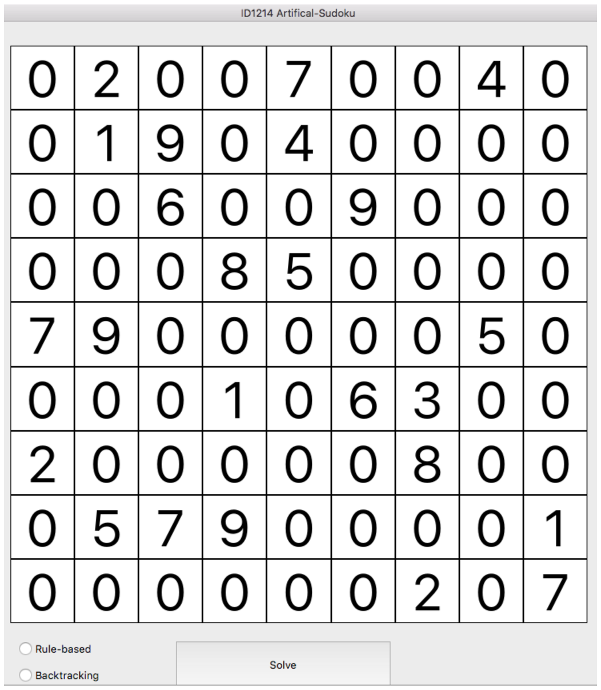
Figure 1: Matrices before and after *nakedPair()*

6 Delivery

We have two kind of representations of our algorithms. One is the GUI based applications that is very easy to use, the other is using the console to generate matrices.

GUI application

Open the application and tick the mode you want, either *Rule-Based* or *Back-tracking* then click Solve to generate a solution. We did not implement a way to


ID1214 Artifical-Sudoku

0	2	0	0	7	0	0	4	0
0	1	9	0	4	0	0	0	0
0	0	6	0	0	9	0	0	0
0	0	0	8	5	0	0	0	0
7	9	0	0	0	0	0	5	0
0	0	0	1	0	6	3	0	0
2	0	0	0	0	0	8	0	0
0	5	7	9	0	0	0	0	1
0	0	0	0	0	0	2	0	7

Figure 2: GUI application

enter a matrix in the GUI application so if you want to test different matrices you have to replace the vector in the code.

```
std::vector<vector<int> > board = {
    { 0, 2, 0, 0, 7, 0, 0, 4, 0 },
    { 0, 1, 9, 0, 4, 0, 0, 0, 0 },
    { 0, 0, 6, 0, 0, 9, 0, 0, 0 },
    { 0, 0, 0, 8, 5, 0, 0, 0, 0 },
    { 7, 9, 0, 0, 0, 0, 0, 5, 0 },
    { 0, 0, 0, 1, 0, 6, 3, 0, 0 },
    { 2, 0, 0, 0, 0, 0, 8, 0, 0 },
    { 0, 5, 7, 9, 0, 0, 0, 0, 1 },
    { 0, 0, 0, 0, 0, 0, 2, 0, 7 },
  };
```

Console approach
Run the program by writing the following in the shell:

```
g++ -std=c++11 Sudoku.cpp -o sod && ./sod
```

This will compile and run the program. To select whether to solve with *Rule-Based* or with *Backtracking* answer the following question with either the character r for *Rule-Based* or b for *Backtracking*:

```
Write what mode Rule−Based or Backtracking write r/b:
```

To change matrix do as with the GUI-application.

7 Conclusion

This was a fun project and we learned a lot about heuristics and the potential in researching more on the Sudoku puzzle as there is so much information on the topic.

References

[1] *http://hodoku.sourceforge.net/en/tech*$_n$*aked.php.* $2018 - 12 - 01$.

[2] *http://hodoku.sourceforge.net/en/tech*$_s$*ingles.php.* $2018 - 12 - 01$.

[3] *http://sudopedia.enjoysudoku.com/Hidden*$_S$*ingle.html.* $2018 - 12 - 01$.

[4] *https://www.technologyreview.com/s/426554/mathematicians-solve-minimum-sudoku-problem/.* 2018-12-01.

[5] *https://www.svd.se/sudoku.* 2018-12-10.

[6] *https://www.telegraph.co.uk/news/science/science-news/9359579/Worlds-hardest-sudoku-can-you-crack-it.html.* 2018-12-10.

[7] *https://www.qt.io/.* 2018-12-12.