C++ FOR FINANCE

ASSIGNMENT 1: SUDOKU SOLVER

Student ID: 2031611

JANUARY 21, 2020

UNIVERSITY OF BIRMINGHAM
Lecturer: Laurence A. Hurst

Introduction

The objective of this assignment was to write a program in C++ 14 to solve the 9x9 Sudoku puzzle shown in Figure 1. Only C++ 14 standard libraries were used to write the program. Four approaches were considered but two were investigated in detail and only one was selected.

A standard Sudoku puzzle consists of 9x9 cells/grids. Each cell must only contain a digit from 1-9, and the following constraints must be satisfied to solve the puzzle:

1. Each row (blue) must contain only an instance of digit.
2. Each column (green) must contain only an instance of digit.
3. Each 3x3 region (red, also called a nonet), must contain only an instance of digit.

The program is required to solve standard (9x9 grids), proper Sudoku puzzle (i.e. puzzle that has a unique solution). This means puzzle has at least 17 clues (solvable) [3] and at least 8 distinct clues [4].

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

Figure 1 shows the Sudoku puzzle to solve [5].

Solving Methods

There are four common methods to solve Sudoku puzzle algorithmically, namely brute-force (with backtracking), stochastic search, constraints programming and exact cover method [1]. Exact cover method with Dancing Links techniques by D. Knuth (2000) was initially considered. Though it was regarded as the most efficient method amongst the four mentioned above, it was not chosen eventually due to its complexity and difficulties to implement [2]. Brute-force with backtracking was the obvious choice because it can provide a simple working solution.

Brute-force search is an algorithm that find all possible solutions to a problem and then return the subset of those solutions that satisfies the explicit constraints given by the problem. Computing all the solutions including infeasible ones is a waste of computational resources. Backtracking is an extension to brute-force method. The essence of backtracking is to solve problems recursively by generating a solution incrementally. At each step, it checks the solution against the constraints of the problem, thereby reducing the size of search-tree. If the solution is true, then it continues generating subsequent solutions. If it is false, it backtracks to the previous solution and computes other solutions.

Algorithm Summary

In the case of Sudoku puzzle, brute-force method essentially tries to fill a digit from 1-9 to each empty cell. Combining with backtracking technique, it does constraints checking at every step. If the constraints are satisfied, it proceeds to the next cell and repeat the same steps. If no suitable candidate is found, it backtracks to the previous cell and repeat the same steps with previously unassigned digits to that particular cell and so perform search within a smaller search space.

The pseudo code is as follows:

1. Find coordinates of empty cell

If there is none, return true

2. For digits from 1 to 9
    a. If there is no rules violation at row, col
        assign digit to row, col and recursively fill in the remaining empty cell.
    b. If recursion successful, return true
    c. else, remove digit and try another
3. If all digits have been tried and still does not work, return false.

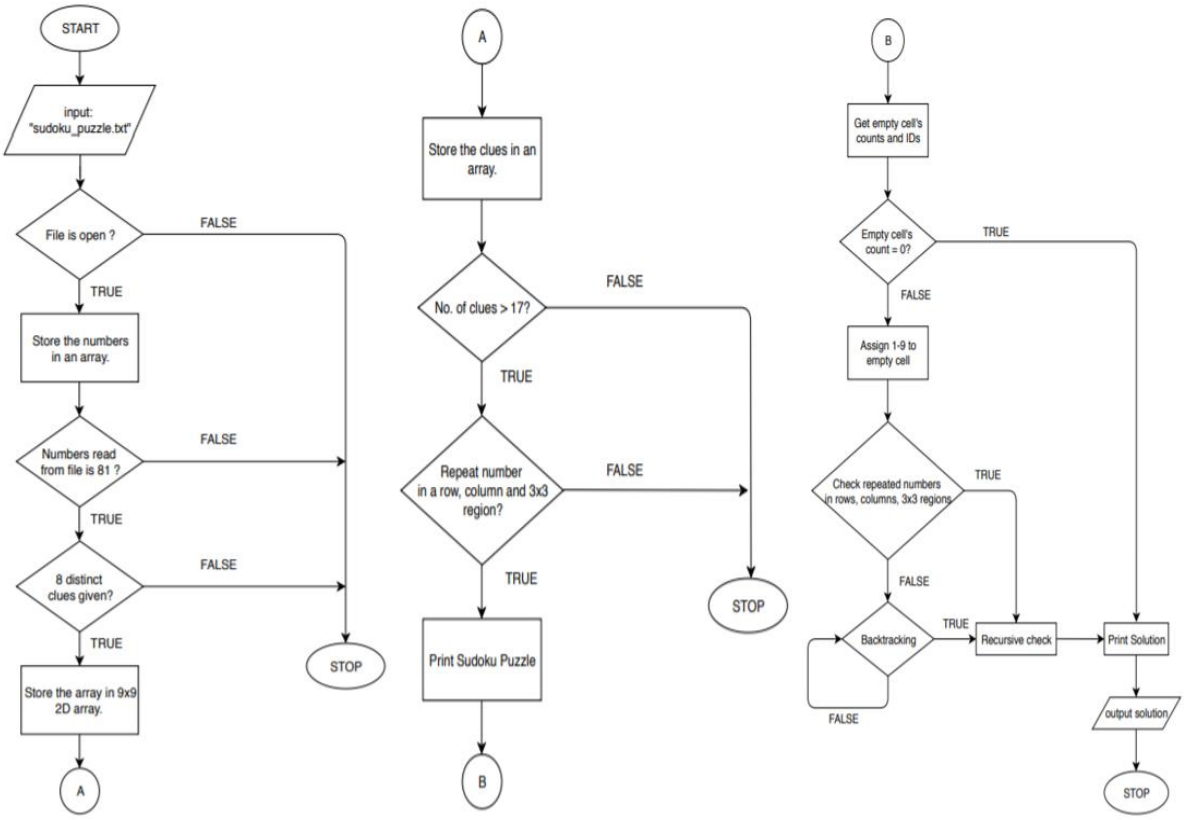

Figure 2 shows the flow chart of Sudoku solver program.

Design-level Efficiency

The brute-force with backtracking method took about *100ms* to run. Exact cover method source code from online source [6][7] was used for comparison. Its modified version (FOR COMPARISON ONLY, NOT TO BE ASSESSED) for solving the above puzzle can be found here [8]. Solving the same puzzle took exact cover method about *23ms*, which is **4** times faster. Solving the sudoku puzzle by nature involves updating and downdating the data structures. The speed advantage of exact cover method over brute-force with backtracking is a result of the use dancing links technique developed by D. Knuth (see [2] for more details). Instead of performing the computationally expensive actions of removing and restoring data, this technique creates a special data structure that hide (cover) and unhide (uncover) data to update and downdate the Sudoku puzzle. All the tests were run on machine with i5-7200U (up to 3.18Hz), 8GB Memory.

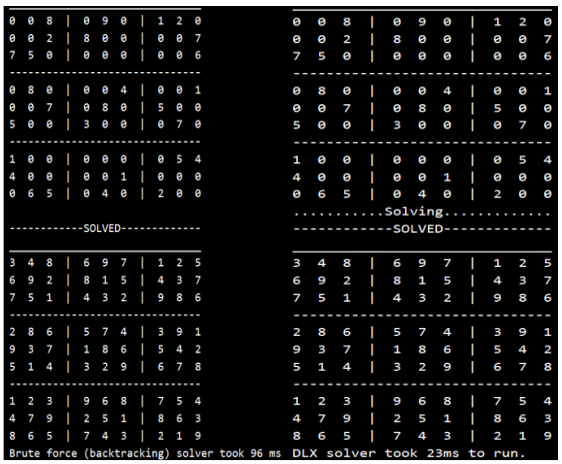

Figure 3 shows the time taken for each method.

Language-level Efficiency

In terms of language-level efficiency, passing arguments by reference into functions were used instead of passing arguments by value wherever possible (see readSudoku.cpp). This way no copies of arguments were created. In theory, it should improve program's efficiency. Comparison tests of passing by reference and value were done. It was observed that the performance gain was trivial. This could be due to the small size of arrays being passed, which in this case were 81 and 9x9 arrays. The efficiency gains of passing by reference would be more significant in the case of passing large arrays. For example, in quantitative finance where it involves analysing large co-variance matrix of portfolios containing hundreds of financial instruments. Nonetheless, it is a good programming practice to follow.

The other programming practice to improve efficiency is choosing pre-fix increment/decrement over post-fix increment/decrement (see line 103 of readSudoku.cpp). In the case of pre-fix, the object just increase/decrease itself without creating a copy.

In this application, the size of data structure will not change throughout the project. Hence, the data containers used were fixed-size arrays rather than vectors. In theory it is faster than using a vector. However, tests results suggested that the difference were negligible. This could be due to the array size was too small for any significant advantage to be observed.

Possible Future Improvements

There are some possible improvements on the program given enough time and C++ experience. In terms of program optimisation, design-level efficiency is far more impactful than language-level efficiency [9]. Thus, exact cover with dancing links technique would be a better choice if one had enough time and proficient enough C++ skills. For comparison purposes, using the code from here, it was observed that this algorithm can solve the anti-brute-force puzzle within similar duration (*23ms*). It would take brute-force method up to 30 seconds to solve [1].

Conclusions

Brute-force method is a good algorithm choice as it can meet all the requirements. Choosing a simple and working solution is important for maintainability, especially when working as a group. Despite that, code must be written with functionality extension in mind. In this case for example, the program can easily be edited to solve Sudoku puzzles of different sizes (see line 8 of readSudoku.hpp).

References

[1] https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

[2] Knuth, D.E. (2000). 'Dancing Links'. Stanford University.

[3] McGuire, G., Tugemann, B. and Civario, G. (2014). There Is No 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration. Experimental Mathematics, 23(2), pp.190-217.

[4] http://pi.math.cornell.edu/~mec/Summer2009/Mahmood/More.html

[5] Hurst, L. A. (2019). "C++ For Finance – Assignment 1", University of Birmingham.

[6] https://www.geeksforgeeks.org/exact-cover-problem-algorithm-x-set-2-implementation-dlx/

[7] https://github.com/KarlHajal/DLX-Sudoku-Solver/blob/master/DLXSudokuSolver.cpp#L30

[8] https://github.com/cko22/Sudoku-solver-DLX-Exact-Cover-.git

[9] Gregoire, Professional C++ pg 472.Wiley (ISBN: 1119421306)