Initial design; Sudoku in C

Algorithms & Data Structures

SET08122

Andrew Dickinson
Matriculation: 40538519
Word | Page count: 1491 | 3
Submission date: 06/03/2023

Introduction

Sudoku is a game loved by many, found in numerous places from newspapers, to books to apps. The game dates back to the 18th century when it was called “Latin Squares” and didn’t get the name Sudoku until 1984 when it appeared in Japan (Easybrain, 2023). There are multiple methods to implement and solve Sudoku puzzles. This report sets out to discuss and design an implementation plan in C for a playable game in full.

This will be my first project in C, and have spent the last few weeks trying to learn the language from scratch. It has been a steep learning curve, but hopefully this will be a fun project to implement.

(Stuart, 2007) mentions three important standards which must be met when creating a good sudoku puzzle, these are:

1. A puzzle should have only one solution.
2. There must be a logical solution to all puzzles (i.e. no guessing and can use clues to solve)
3. Puzzles should have some kind of symmetry (of the starting clues), usually around the centre.

All of these will add further complexity to the project, however the design will strive to achieve them where possible. (Stuart, 2007) also says that the average number of clues is 28, and never more than 32, with a possible low of 17. He goes on to say that a puzzle must contain at least 8 of the 9 possible numbers as clues – this is to avoid swapping of solution numbers (hence leading to multiple solutions).

(Berggren, P., Nilsson, D., n.d.) discusses and analyses three different algorithms – backtrack, rule-based and Bolzmann machines. They conclude that rule-based (a combination of heuristic and a brute-force algorithm) is the best performing, with the backtracking coming in second. These both have pros and cons, implementing a rule-based algorithm will possibly require more time than this project has. This can be seen by the number of strategies implemented by (Zamon, G., 2015). Backtracking on the other hand may take an unacceptable time to compute.

(GeeksforGeeks, 2023) outlines different implementations for a sudoku solver suggesting the best performing is cross-hatching with backtracking.

(Harrysson, M., Laestander, H., 2014) suggest that DLX (Dancing Links technique) can be used to solve Sudoku puzzles very efficiently. Known as Algorithm X, and developed by Donald Knuth, it is a depth-first, backtracking algorithm using doubly-linked circular lists (Knuth, D., 2019). It is used to solve the exact cover problem (such as Sudoku is). This was a large area of research for this project, however there has not been enough time to fully comprehend how to implement this algorithm. As such, this will not be attempted to be used unless time allows, but is certainly an area of interest for future reading.

The project will focus on building a minimum viable working version of Sudoku loosely based on Python examples as seen in (Medium, 2021) and (Lvngd, 2020).

The main project aims are as follows:

- To have an initial menu – to include difficulty settings / start new game.
- Generate and display game board – this should be checked ensure game is valid and solvable.
- Take inputs from the user – these should be checked to ensure valid inputs (1-9).
- Recognize a completed game – At this point, the user should be notified as to if their solution is correct or not.

Additionally, the following features are aimed as time allows:

- The ability to undo / redo – To allow the user to undo previous moves, potentially then redoing them.
- Save / continue games – To facilitate exiting the program and continuing at a later time.
- Play against the clock – To add an extra difficulty for the user.
- Record history of play – This should then be able to be played back so the moves of a game can be watched.
- Differing board sizes – Increase/Decrease number of cells in a board to have variations of games.

Design

Firstly, a game board (grid), will be stored as a 2D array. This will allow fast access to elements by reference of row and column indexes (0 indexed). Arrays are constant access time O(1), with O(n) time for insertion/delete. The grid will be initialized with zeros initially, indicating an empty space (alternatively, a space may be used to make the grid more readable with the grid being a 2D char array). This will require iterating through the array (O(n)) time). The grid will be able to be printed to the console in a similar manner.

Following this, the grid will be partially filled by randomly completing boxes 0, 4 and 8 (for standard 9x9 grid). These boxes do not interfere with one another in terms of the constraints of Sudoku (1-9 must appear exactly once in each row, column and box (standard 3x3)), allowing board generation time to be reduced. Box numbers can be determined by calculating:

```
row_index / num_box_rows * num_box_rows + col_index / num_box_cols
```

To finish generating a completed board, a brute force backtracking algorithm will be used. To set this up, the grid is iterated through to find all empty cells, for each found, a node for a doubly linked list will be created (doubly linked allows backtracking). This will store the row and column values for each empty space (and pointing to the next and previous empty space), reducing the need to iterate through the entire grid later to locate empty spaces. Once all empty cells have been found, a function (complete_board) will be called. The first empty space is obtained and a random value 1-9 (for the standard board) will be generated (random; ensuring that each game generation is different). A check_valid function will then be used to check the generated value against its intended row, column and box to check the value has not already been used. If the value is valid, it will

be inserted into the grid, and the pointer to the list of empty cells will be updated to point to the next available cell. Another function (solve) will then try to solve the puzzle. It will do this in a similar manner, first the list of empties is checked it is not NULL (if it is, the board is complete, and can be returned). A for loop will be used to iterate through the possibilities for each cell. For every iteration, it will check if the value is valid in the selected cell. The value will then be inserted, the empties list updated, and the function recursively called from an if statement until the list of empties is NULL. For each recursive call, there is the option to reset the grid value and empties list in the else statement . This pattern is similar to that mentioned by (Lvngd, 2020), *choose -> explore -> unchoose.*

To generate the puzzle board, the completed board will be copied. As earlier, the corner and centre boxes do not affect each other, and so are the easiest places to start removing values. The difficulty rating will be based on the number of values removed from a completed puzzle. The program will aim to remove 49 for easy (32 clues), 53 for medium (28 clues), and 61 for hard (20 clues), though these will be set to time out if removing numbers is taking too long. The first 12 values will be removed from the corner and centre boxes to speed up the process. After this, the value to be removed will be random. When a value is removed, the puzzle will be checked it can be solved in the same way as above, however this time, counting the number of solutions so as to check there is only one. Each value removed will be added to a stack containing arrays for the row, column and value. This will allow easy retrieval to backtrack should there be more than one solution.

Once the puzzle has been created, gameplay can begin. The user will be requested to input the coordinates of the cell to update along with the value. To allow undo and redo moves, each users move will be added to a stack as above. When a move is undone, it will be added to a different stack to allow redo. When a new move is played, the redo stack will be initialised to NULL.

To allow saving a game / continuing and or replaying the moves of a game, a copy of the puzzle board will be kept. This can be iterated over and each value can be saved directly to file. Each move a player makes can be added to a queue. When a user selects to save a game, both the original puzzle and users moves can be written to file.

There is certainly areas in this project (especially the solving algorithm) which could be improved for efficiency. These will be continued to be reviewed and researched in an effort to create a program where the Sudoku puzzle is generated in a reasonable time (<20 seconds – ideally a lot less).

References

1. Berggren, P., Nilsson, D. (n.d.). *A Study of Sudoku solving algorithms.* 3rd Feb 2023, https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group6Alexander/final/Patrik_Berggren_David_Nilsson.report.pdf
2. Easybrain. (2023). *The History of Sudoku.* 11th Feb 2023, https://sudoku.com/how-to-play/the-history-of-sudoku/
3. GeeksforGeeks. (2023). *Sudoku|Backtracking-7.* 3rd Feb 2023, https://www.geeksforgeeks.org/sudoku-backtracking-7/
4. Harrysson, M., Laestander, H. (2014). *Solving Sudoku efficiently with Dancing Links.* 22nd March 2023, https://www.kth.se/social/files/58861771f276547fe1dbf8d1/HLaestanderMHarrysson_dkand14.pdf
5. Knuth, D. (2019). *The Art of Computer Programming.* 20th Feb 2023, https://www.inf.ufrgs.br/~mrpritt/lib/exe/fetch.php?media=inf5504:7.2.2.1-dancing_links.pdf
6. Lvngd. (2020). Generating and solving Sudoku puzzles with Python, 21st Feb 2023, https://lvngd.com/blog/generating-and-solving-sudoku-puzzles-python/
7. Medium. (2021). Building a Sudoku Solver and Generator in Python, 22nd Feb 2023, https://medium.com/codex/building-a-sudoku-solver-and-generator-in-python-1-3-f29d3ede6b23
8. Stuart, A. (2007). *Sudoku Creation and Grading.* 2nd Feb 2023, https://www.sudokuwiki.org/Sudoku_Creation_and_Grading.pdf
9. Zamon, G. (2015). *Sudoku Programming with C.* 6th Feb 2023, https://learning.oreilly.com/library/view/sudoku-programming-with/9781484209950/9781484209967_Ch02.xhtml#Sec1