Algorithms and Data Structures - Assignment 1

Introduction

Sudoku is a puzzle game that involves filling a $n \times n$ grid with digits so that each column, each row, and each of the n boxes contains all of the digits from 1 to n. We define a state by the i-jth cell in the sudoku (e.g. a cell at 1, 2) and the action by adding a value to cells containing zeros (empty cells). In this project, we developed a Python program to solve Sudoku puzzles using both exhaustive search and backtracking algorithms.

Exhaustive Search and Backtracking

An exhaustive search algorithm is an algorithm that for every search space explores all states in a systematic manner and then filters out the correct solution(s). Backtracking, on the other hand is a way of designing an algorithm that already checks if the next state we want to add/modify is correct - if yes then we proceed to the next one, but if not then we go back to the previous state. Such an approach allows us to only go in the direction of the correct solution and makes the search tree significantly smaller.

For the case of our sudoku algorithm if we set the backtracking parameter to false then we add all posible combinations of values to the sudoku and only if it is full we check if it is correct. That requires a lot of resources and is inefficient. If we want to change this to backtracking then we can simply check if the value we want to add to the next cell is correct meaning that it is not already placed in the same row, column and box as the current cell. Finally the algorithm ends by checking if the entire sudoku is correct.

Complexity and search trees

Exhaustive search The worst case time complexity of exhaustive search for an n-sized grid is:

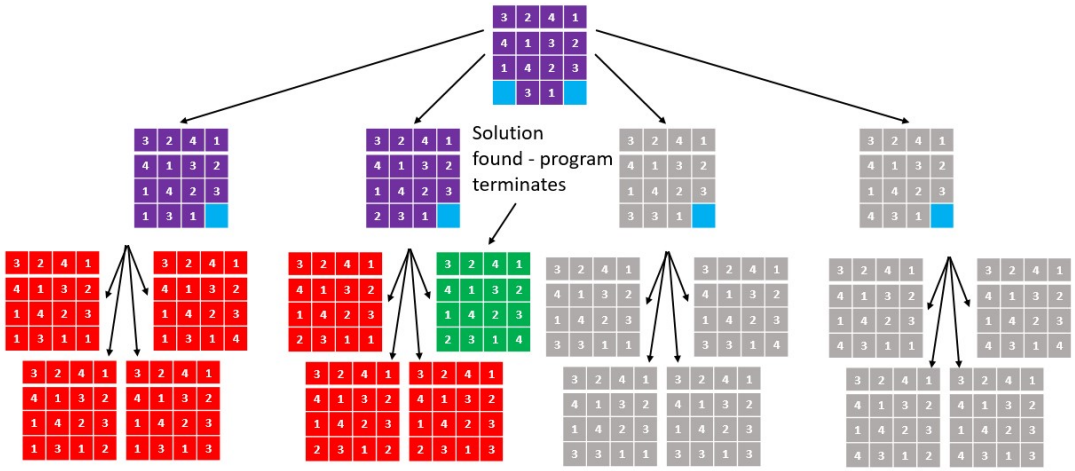
$$O(n^{n^2}) \tag{1}$$

as we have n numbers to place in every cell and the number of times we have to do this is the size of the grid so n^2 for a square sudoku grid.

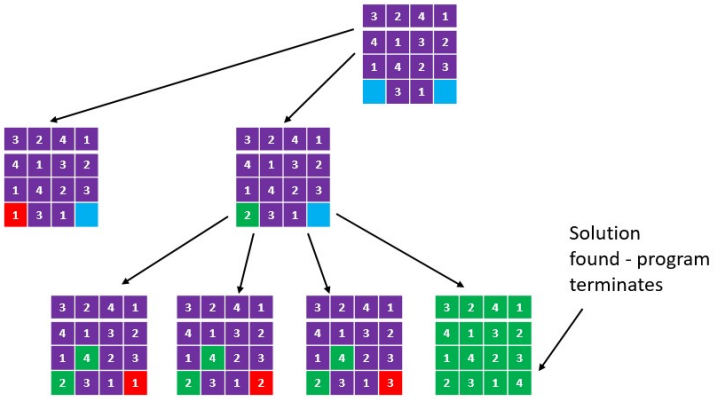
Backtracking While the backtracking algorithm also explores the possibilities of placing n numbers in each cell, it includes an optimization step. Before placing a number, the algorithm checks whether it leads to a valid solution. If a number leads to an invalid state, the algorithm backtracks and tries a different number. This reduces the total number of configurations that need to be checked. The worst-case time complexity is still the same, because we would need to explore all possible choices.

Search trees To illustrate the performance of both algorithms we will now look at the corresponding search trees for a 4 x 4 sudoku with two empty cells:

Exhaustive Search Algorithm Tree



Backtracking Algorithm Search Tree



The first figure that shows the search tree for an exhaustive search approach proves that we have to explore every possible value and thus end up with many duplicate values in a sudoku that are computed unnecessarily. As it can be observed in the second figure, backtracking determines that adding 1 is incorrect, and even though there are empty cells it goes back to the previous state and chooses 2 which leads to then finding the correct solution by finally filling the last cell with 4. In general this approach allows the computer to only explore the branches of the search tree which lead to the solution and skip the incorrect branches, so the search tree does not grow exponentially in size as in the case of exhaustive search. Furthermore, because of this property, backtracking is optimal (guaranteed to find a solution if a sudoku can be solved).

Possible improvements

One idea is to utilize the advantage of backtracking by rotating the sudoku in a way that we have the highest count of filled cells at the top-left part of the grid, which allows us to discard more incorrect numbers at the beginning and leads to a faster computation of the algorithm. The more we know at the beginning, the less computations are needed at the end, as the algorithm can rapidly eliminate incorrect paths, reducing the overall search space.

Size of the search space for an n x n sudoku

The search space for an n x n Sudoku by exhaustive search equals $O(n^{n^2})$, as every cell in the n x n grid can be filled with any of the n numbers. For a $9x9$ Sudoku grid, this becomes 9^{9^2} or 9^{81}. Now, we perform the calculation:

$$9^{81} \approx 1.97 \times 10^{77} \tag{2}$$

An average computer performs about 10^7 operations per second in python. When we divide this number by the size of the search space we obtain the time the program would take to execute in seconds:

$$\frac{1.97 \times 10^{77}}{10^7} = 1.97 \times 10^{70} \tag{3}$$

For comparison, the number of seconds in a year is approximately 3.16×10^7 Hence, running an exhaustive search on an empty 9 x 9 grid to get a solution in real time is practically impossible. This underlines the need for more efficient algorithms like backtracking.

Summary and Discussion

After testing both algorithms, we found that backtracking is more efficient than exhaustive search for solving the sudoku, because it allows us to skip parts of the search tree that do not lead to finding a solution. On average, this makes the algorithm much faster, but in the worst case scenario the time complexity is the same as for the exhaustive search approach.

Secondly, even though both algorithms allow us to solve relatively small sudokus (of size $n \leq 16$ for backtracking and $n \leq 4$ for exhaustive search), for growing sizes the number of computations required to obtain the solution grows super exponentially so these algorithms are not applicable for any n-sized sudoku.

Contributions

Mateusz Wilk (4003748):	Methods implemented: set_grid, get_row, get_col, get_box Exhaustive search design Backtracking code implementation Report introduction Report paragraph with description of exhaustive search and backtracking Report search tree paragraph Report formatting and spell checking Report conclusion and summary
Koorosh Komeilizadeh (3893995):	Methods implemented: Sudoku initializing, is_set_correct, check_cell, check_sudoku Exhaustive search code implementation Backtracking design Report complexity paragraph Report possible improvements paragraph Report search space for n x n sudoku paragraph Report search trees visualization