Backtracking

Our problem is to solve a 9x9 Sudoku puzzle using the backtracking algorithm. A Sudoku puzzle is a 9x9 grid with clues placed strategically on the grid so that it may be solved. To solve the Sudoku puzzle there may be no repeats of the numbers between 1 and 9 in each row, column and 3x3 box. There are obviously many ways to solve a Sudoku puzzle, one being a brute force approach, where one generates all possible configurations of the numbers between 1 and 9 for every empty cell until the correct solution is found. This is however a very naive approach and thus not very good as it is nowhere near optimal. Thus we consider the much better backtracking algorithm for solving a Sudoku puzzle.

In the backtracking algorithm, we solve is by making use of a choice of which number between 1 and 9 to put in a cell subject to the constraints of Sudoku puzzles in which there may be no repeating number in any row, column or 3x3 box. Thus, to achieve our goal of finding a valid solution to a Sudoku puzzle, we sequentially find empty cells and for that cell find a suitable number between 1 and 9 that satisfies the constraints. If a suitable number is found we move to the next empty cell and repeat. If a cell is found in which the constraints are not satisfied for any number between 1 and 9, we “backtrack”, by moving to the previous cell and attempting to find another suitable number for that cell, if one is found we continue. If one cannot be found for the previous cell, we must “backtrack” even further, attempting to find different suitable numbers which satisfy the constraints until a full solution to that row is found. Once this occurs we repeat this process sequentially for every row until either a solution to the Sudoku puzzle is found or it is deemed that no solution exists and all possibilities have been exhausted.

Problem: Given a 9x9 partially solved grid, find a solution using the backtracking algorithm.

In pseudo-code we can view our strategy as:

Input: a 9x9 grid stored as a 2-dimensional array.

Output: Solved 9x9 grid/ True if board is solved and False if board cannot be solved.

- sequentially find row, column of unassigned cell

- if there is none, return True

- for numbers from 1 to 9 do

- if number is possible subject to constraints

- assign digit to cell and recursively try fill all other cells

- if recursion successful, return true

- else, remove number and try another

- if all numbers tried for all cells and nothing successful, return false

Proof of Correctness:

Direct Proof:

We are given a partially solved 9x9 grid. We sequentially traverse this grid row by row by means of a double for-loop. Thus, for each cell in a row we consider the case if the cell is empty. I.e. if grid[row][col] == 0. If the cell is indeed empty, we must now consider all the values starting 1 and ending at 9 for that empty cell by means of another for-loop. Thus far it has been very trivial and all is correct. In order to find the correct number for the cell we make a function call to isPossible() to check the numbers between 1 and 9 for the first number which successfully satisfies all 3 of our constraints. The first value to be passed is the number 1. Within the function, the row is iterated over by means of a for-loop and checked by means of an if-statement to make sure the row does not include any other 1’s. If it fails, isPossible() will return False and the number will be incremented to 2. If there are however no other 1’s in the row, we similarly iterate over the column and check that there are no other 1’s and return false if there is, otherwise we continue to our final constraint. In our final constraint we must consider the 3x3 block to make sure there aren’t any 1’s within that block. We determine which block we are working in, as well as the starting index of that block via simple math(int initCol = (col/3)\*3; int initRow = (row/3)\*3). And we iterate over this block only via a double for-loop. If there is indeed a 1 in the block, then isPossible() will once again return False. If there is no other 1 in the block, we have successfully passed all 3 of our constraints and the function will return True, indicating that a 1 may be placed in that empty cell. The 1 is placed within the empty cell(grid[row][col] = n, which is 1 at the moment). We then test if the board is solved by recursively making a call to solve within our if-statement. If it is the only cell and we have successfully filled it, the function will return true. If this is not the case, our function will move to the next cell and repeat the process described above. (call solve() to check, as well as if it is the correct number in the correct position. Since this is the only cell we have filled in, and it is not the only cell on the board) This process is continued until a cell is reached in which no numbers between 1 and 9 successfully preserve all of our constraints within isPossible. When this occurs, our if-statement fails and we instead backtrack to the previous cell(grid[row][col] = 0) and try find a suitable number to replace the 1 that was previously there, for example 2.

Thus our algorithm successfully solved the Sudoku puzzle.

Complexity Analysis:

For our complexity analysis we use n=3, resulting in n^2 = 9.

Space Complexity:

The standard representation of a Sudoku puzzle is an n^2 x n^2. This grid is divided into n^2 boxes each of size n x n. Thus our memory complexity is O(n^2 x n^2) to store the 2-dimensional array needed

Time Complexity:

In terms of the complexity of our backtracking algorithm, a good description for the amount of work done is (a combination of) the comparison for a cell, the numbers 1 and 9 with the corresponding row, column and block. The assigning and removal of numbers in cells is completed in constant time and therefore does not affect the complexity.

In the best case the grid is essentially fully completed and there are minimal comparisons required and the only operation done is the traversal of the 2-dimensional array in search of an empty cell.

In the worst case all the cells are empty. The cost of determining if a number for a certain cell is possible for the entire grid is 3n^4. This is evident as for an empty cell, in the worst possible case, that number will be compared with an entire row which is n^2 length, and entire column which is also n^2 length and an entire block which is also n^2 in size. This results in n^2 + n^2 + n^2 = 3n^2 for checking if that number satisfies our 3 constraints. This needs to then further be completed for the entire grid which is n^2 x n^2 resulting in 3n^2\*n^2 x n^2 = 3n^6.

It can further be seen that the biggest term in our complexity function for the worst case is 9^(n^2 x n^2). For the first empty cell there are n^2 possibilities that may be chosen for that block. For the second empty cell there are n^2 possibilities FOR EACH of the n^2 possibilities of the first empty cell. If there are 3 empty cells there are n^2 possibilities for the first cell, and for each of those possibilities there will be n^4 possibilities for the other two empty cells. This means for the choice of numbers between 1 and 9 and extrapolating this concept for the entire grid we find there are 9^(n^2xn^2) possibilities.

Thus our complexity function is: gw(n) = 9^(n^2xn^2) + 3n^6 + 1 + 1, representing the number of possibilities, whether a number is possible and satisfies all constraints, and the assigning and removal of numbers in cells respectively. Thus the backtracking algorithm in the worst case is O(9^(n^2xn^2)).