Sudoku

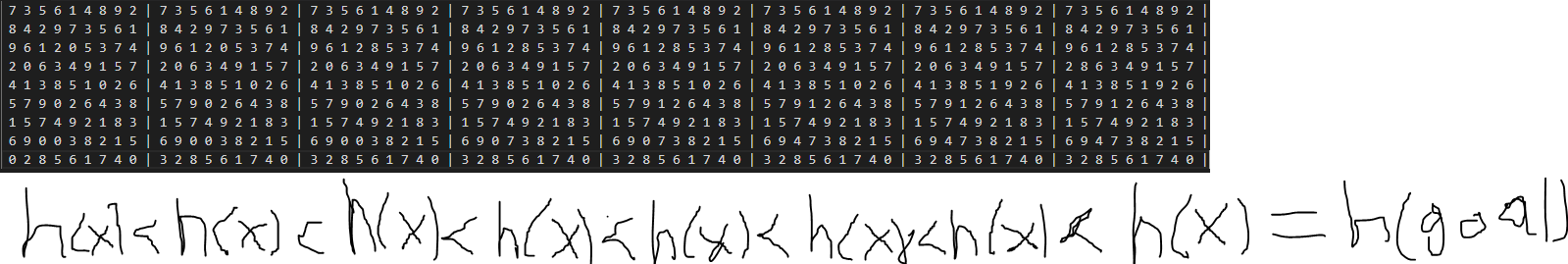
The Goal of Sudoku is to fill in a nine by nine grid with the numbers 1-9 so that every column, row, and darkened 3x3 box so that they all have exactly one digit 1-9. Other symbols can be used instead of numbers, but there must be nine of them. A good strategy when solving Sudoku puzzles on paper is to find the empty square with as few options as possible. Usually there is one square that has only one option. Fill it in. Find another. Fill it in, until the board is clear.

In my first attempt at solving the problem I viewed the problem as a constraint satisfaction problem. In my Second attempt I tried a randomized algorithm. Both attempts were failures java arrays not working like I thought they were. Should have used Python.

In terms of data, the only required data is the initial state of the Sudoku grid, and the rules of Sudoku. Once that is given the validness of the board, for the heuristic function, needs to be calculated. In the case of the constraint satisfaction problem, the number of digits that are allowed for each of the remaining free spaces needs to be determined. I had been planning of designing a heuristic for ordering the digits tried in the spaces. It likely would have had something to do with the number of times the digit was currently on the grid.

The complexity of the randomized algorithm would very between runs, even with the same input. Its average run time would very base on parameters. The more digits predetermined would shorten the run time. For example, if say all but three fields are filled in the random algorithm will solve the puzzle fairly quickly. If one the example puzzles is used as the start state, I am not sure it is possible for the puzzle find the correct solution. Since the heuristic comparison is the problem, it would be faster to use a liner search of the possible ways to fill-in the blank squares with the relevant symbols.

In both algorithms, I convert the text formatted puzzle to a 2-diminsinal array (string for the CSP[][] and int[][] for the RA). In the randomized algorithm I subtract 1 from all digits, and swap all the negative ones with zero. In the original problem, this would be like filling in all the blank squares with ones with a pencil. The numbers in the original data are labeled as fixed. Then a pair of three random numbers between 0 and 8 are selected. For a row, column and value. If the row and column are fixed, than new numbers are selected. Otherwise the random value is the new value in the random bank square. In theory if the new grid had a worse heuristic score than the previous value the new value would be thrown out. I define my heuristic is the sum of three different numbers which are three other values. One for rows, columns and what I labeled as cells, but the 3x3 boxes. The three scores were defined by how many of the 27 classes had of each of the digits. Nine for example would return a score of zero, but 1 would have as score of 8. So to reiterate, add up the scores for each of the digits in each of the 27 classes, then add up the 27 sums, and that is my heuristic function. I have some commented out code that would cause the search to reset after a certain number of generations, in case the search became stuck on a local minimum of the heuristic function. Most of what I had done for the SCP was making a data structures to keep the time cost down.



In conclusion it was a bit of train wreck, apparently I think I must have read my calendar wrong. Had I better utilized the time I would have solved the problems I was having. The data keeping up with all the data was hard. I had forgotten quite a bit about java since my earlier classes, which focused more on java. My randomized algorithm would likely not likely hard time constraints well as the constraint satisfaction algorithm.

References

https://anysudokusolver.com/images/Sudoku-Solved.png