Sudoku Solution Report

Difficulty of approach:

* I have developed a type of depth-first search algorithm with a constraint satisfaction. I selected this search algorithm since the number of solutions vastly increases as the complexity of a Sudoku increases.
* The primary goal was to minimise backtracking in the algorithm. This led to the decision against using a breadth-first search algorithm. A breadth-first approach would generate a node for every missing value at the initial step, resulting in significant time and resource consumption due how quick and large the space complexity grows.
* An uninformed search was also not considered due to the immense complexity and vast number of possible combinations at each step. Such an approach would likely lead to the search getting trapped in a local minimum, making it highly improbable to find a solution. Considering that hard Sudoku puzzles often start with more than 50 initial cells to solve, this approach quickly becomes unfeasible.
* As for heuristics, the only viable option seemed to be the number of remaining solutions. However, this proved to be an ineffective heuristic, as even a small number of remaining solutions could lead to a local minimum, not aiding in improving the search results. Therefore, I did not opt for a greedy or A\* search algorithm, as these methods are not memory-efficient for this particular task.

Description of algorithm:

* My algorithm searches through all the remaining empty cells. For each cell, it calculates all the possible solutions that cell can take. The solutions are constrained using Sudoku puzzle logic. The three constraints state that no duplicate numbers can be present in a row, in a column or in a 3x3 grid. If this is the case, then the list of solutions that adhere to these constrains are returned. Out of all the cells, the cell with the least number of possible solutions is explored first. This solution constraint that is applied reduces the branching factor to minimise the space complexity.
* Once a constraint-based solution is selected, it is explored from the current Sudoku state, via a depth-first search algorithm.
* If the cell selected for search has more than one potential solution, then a node is created. This allows the algorithm to backtrack to this state to search the other potential solutions of that cell. The algorithm would backtrack, if the current cell solution tree did not yield a complete puzzle.
* For the algorithm to complete a puzzle, a solution that satisfies all three Sudoku constraints must be found for each remining empty cell. If no solution is found prior to there being no remining empty cells, then the algorithm backtracks its frontier state to an earlier node. The node just explored is then removed from the list of nodes to explore.
* The node the frontier decides to revert to is selected by taking a node 75% away from the frontier. This selection was made to optimise how the algorithm searches all the nodes, based off the structure of the Sudoku puzzle search tree.
* My form of depth first constraint satisfaction algorithm is complete. That is, the frontier will search every possible node. After the algorithm backtracks enough where there are no more nodes to search, then it knows that there is no possible solution to the Sudoku puzzle given.

Optimisations and complexity:

* One key optimisation I made which led to a 50% gain in calculation time is that the algorithm calculates the all the possible combinations that can be placed in each row, column or grid once and stores them for each state. When cycling through each unsolved solution, I then only need to use the cell’s location data to gather all available solutions for that cell, rather than re-calculate the options each time a new cell is evaluated.
* Another optimisation made was finding the cells available options across the nine grids using manual list construction rather that the more resource heavy *.flatten()* and *.tolist()* Numpy operations.
* After I calculate all the possible valid options that all the remaining empty cells can take, I take the cell with the minimum number of solutions and explore that space first. This is reduces the space complexity of the algorithm.
* A graph showing a line

  Description automatically generatedMy depth first constraint satisfaction algorithm was optimised through experimentation with how the algorithm backtracks after a node is explored. Through many tests, I found that a trade-off between space and time complexity was the optimum setting, which applied the logic to backtrack my frontier to the first quartile of its node tree each time a node is fully explored.
* The graph across compares some varying choices in the logic of how the frontier backtracks. We can see that there is a 1000% difference between the space complexity of the 75% node backtrack and the 90% backtrack. Then although the space complexity is low in the last node backtrack, the time complexity is 900% larger compared to the 75% backtrack.

Reflections and suggestions for further work:

* I experimented with many different optimisations that all failed in speeding up my solution. A key element I looked to add to my algorithm was multiple forms of stochasticity, after reading that this can optimise some search algorithms. I faced issues with this speeding up my algorithm. One example includes my attempt to select a random solution out of a list of solutions that had the same shortest length. The calculation needed to select a solution sped up my calculation time. Instead, I hardcoded the algorithm to select the first item form the minimum solution list.
* A screen shot of a computer

  Description automatically generatedFurther work would be focused on speeding up the calling and the calculation time of *sorted\_to\_solve()* function. To reduce the number of calls to *sorted\_to\_solve()* I could implement more advanced solution elimination logic to reduce the solution space, such as the Sudoku elimination strategies *hidden pairs* or the X*-wings.*