Analysis of All Strategies

**Goals**

The main goal of this assignment was to utilize my understanding of greedy, divide & conquer, and dynamic programming algorithms to solve sudoku puzzles. For my main project, I created a divide & conquer and dynamic programming solution. For the extra credit, I also created a greedy algorithm.

Computer Specifications

Computer Model: Custom Build

OS: Windows 11

Software Information: Python 3.11 – VSCode

CPU: AMD Ryzen 7 7800X3D 8-Cores 16-Logical Processors @ 5.00 GHz

Memory: 32 GB RAM

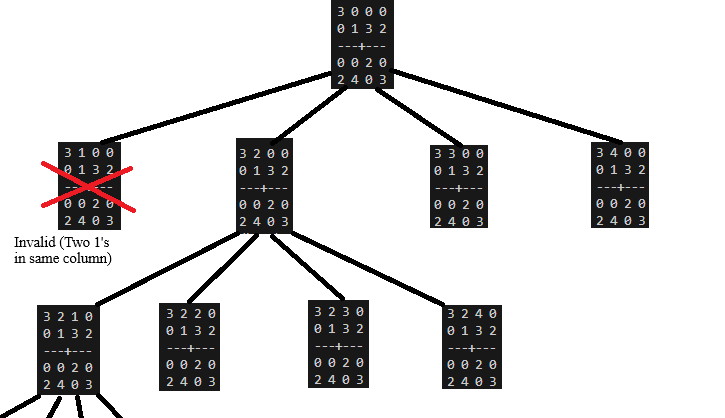
**Algorithms**

**Greedy**

My first algorithm was a greedy algorithm. This algorithm processes inputs by assuming the local optimal solution is the best for each location within the sudoku table (left to right + top to bottom). It goes through each grid location and assigns the lowest possible valid number to that grid location. Because this is a greedy algorithm, it commits to that decision without backtracking and will often not be able to complete the sudoku problem because an early decision results in later grid spaces having no possible value. The time complexity of this algorithm is O(N3) where N is the length of each row within the sudoku grid and N is the number of possible characters within the domain of the solution. This is because the algorithm loops through NxN grid spaces and checks each one a maximum of 3N times to ensure rules are obeyed. The space complexity is O(N2) for the NxN grid.

**Divide & Conquer**

My divide & conquer approach divides the problem into N subproblems and solves one at a time going as deep as possible then backtracking when no values work.



*Figure 1: Visual of Divide & Conquer*

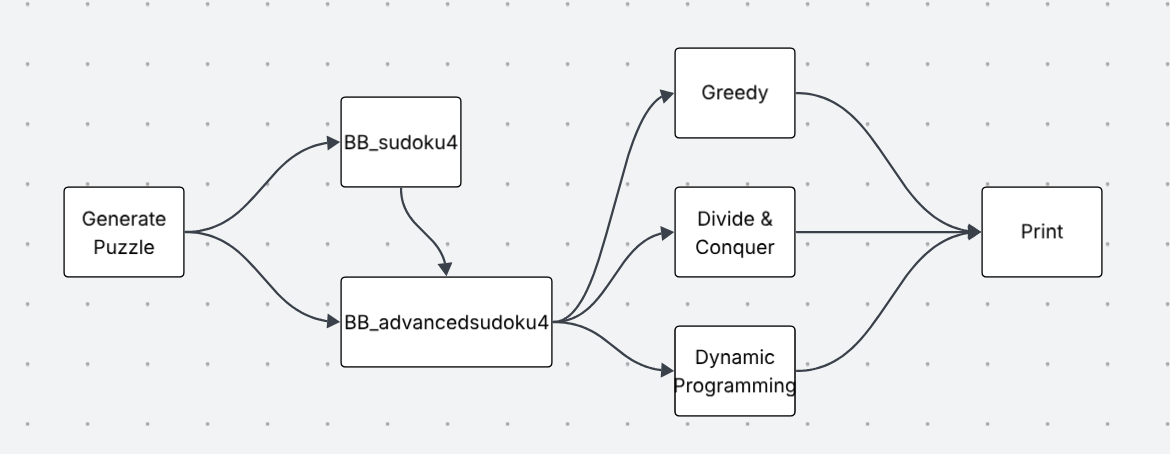
This approach is O(N!) in the worst-case scenario where N is the number of spaces that are blank (spaces with ‘0’). This is because each of the N blank spaces branches out to N new trees that then each branch N-1 times. This results in a rare but worse case of N! order operations. The space complexity is O(N2) where N is the size of each row. This is because the memory required scales with the size of the NxN matrix that must be stored.

**Dynamic Programming**

My dynamic programming approach writes the possible domain of valid values for each space then iteratively fills in values that only have a domain size of 1 and continues eliminating elements from domains of other cells. By storing the possible values within each cell, we are storing overlapping subproblems so cells do not need to be looked at again but instead can be referenced and updated based on changes. Assuming N2 total blank locations and N possible elements within the domain, the worst-case time to update each element of the domain is N3 even though it will never be this bad because elements are filled in at the start. In addition, each removal needs to update domain possibilities from its row, column, and block which is O(N). In total, this algorithm is O(N4) because of all of these factors combined. The worst-case space complexity requires N2 cells with N possible elements in their domain. This means the space complexity is O(N3).

**Program**

At a high level, all BB\_sudoku4 is doing is calling BB\_advancedsudoku4 with a fixed N size of 9. Then BB\_sudoku4 uses the difficulty input to just print what the difficulty was. This is because the sudoku puzzle that needs solving must already be easy, medium, or hard when input. BB\_advancedsudoku4 takes in an input puzzle, a puzzle size, and what method should be used in solving the problem. It then sends the unsolved puzzle S to the chosen method C and returns the result as a 2D list of strings (each string is a character in the domain). Additionally, there are functions to print a given puzzle and to create an unsolved puzzle of difficulty easy, medium, or hard from scratch to be used for testing.



*Figure 2: Block Diagram*

Pseudocode:

FUNCTION BB\_advancedsudoku4(S, N, C):

Symbols <- get\_symbols\_in\_domain(N)

Grid <- deep copy of S

B <- sqrt(N)

FUNCTION find\_empty(g):

FOR each row i from 0 to N-1:

FOR each col j from 0 to N-1:

IF g[i][j]==’0’: return (i,j)

Return none

FUNCTION is\_valid(g, r, c, v):

FOR k from 0 to N-1

IF value is not valid: return False

Return True

FUNCTION greedy(g):

FOR each blank cell (i,j):

Assign first valid symbol

If nothing fits: return (g, False)

Return (g, True)

FUNCTION dac(g):

Pos <- find\_empty(g)

IF pos is None: return True

FOR each symbol v in symbols:

IF is\_valid: set g[pos]=v, recurse, undo if failed

Return False

FUNCTION dp(g):

Init dom[(i,j)] = symbols or given element

While domain doesn’t change

FOR each domain of size 1, remove value from domain in row/col/block

Write values we know

IF grid is solved: return True

ELSE: return dac(g)

SWITCH C:

Pick correct method of solving

Return R

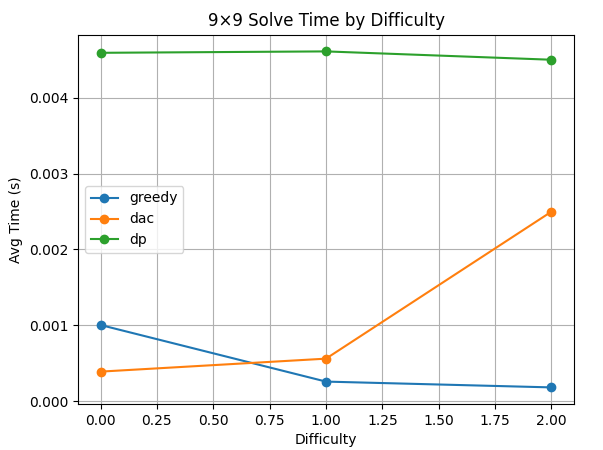
FUNCTION BB\_sudoku4(S, D, C):

BB\_advancedsudoku4(S, N=9, C)

Return R

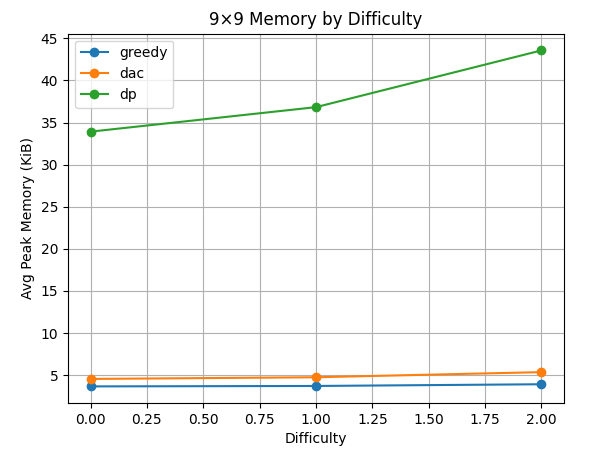
**Experiment 1**

In experiment 1, I compared the time in seconds and the memory in KB of each algorithm as the difficulty of the initial matrix increased (more blank spaces were added). During this experiment, a constant N = 9 was used meaning the sudoku grid was 9x9.



*Figure 3: Time vs Difficulty*

From this graph, we can see how the dynamic programming approach stays very constant throughout different difficulties. This is because the runtime does not depend on how many blank spaces need to be looped through. The greedy algorithm looks as if the runtime improves as the puzzle gets harder, but this downward trend is caused because the greedy algorithm gives up earlier with harder grids. Because it gives up earlier, its runtime is shorter. More about how it gives up will be explained in experiment #3. The divide & conquer algorithm’s time complexity is directly related to the number of blank spaces in the grid that must be solved for. Because of this we see the time to solve increase on the order of N! just as theoretically stated by the time complexity.

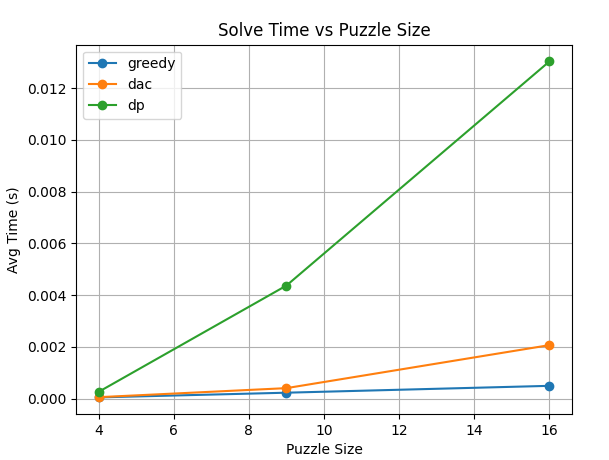


*Figure 4: Memory vs Difficulty*

From this graph we directly what was theorized by the space complexities of all three algorithms. Both dac and greedy scale at a rate of O(N2) because of the size of the sudoku matrix. Dynamic programming is increasing at a much faster rate as seen by its O(N3) space complexity. This is because dp keeps track of a maximum of N elements at each grid location (NxN) on the sudoku puzzle as opposed to only 1 in the other algorithms.

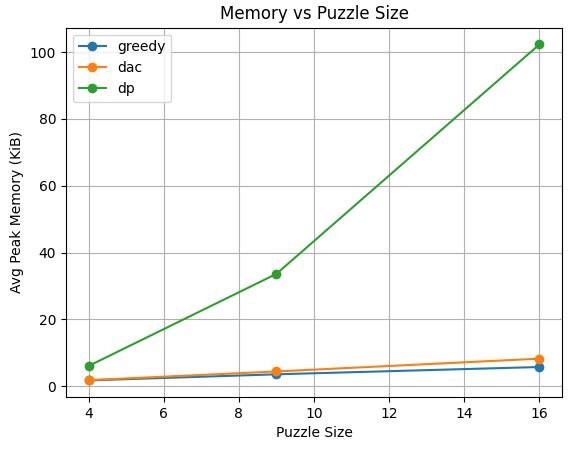
Experiment 2

In this experiment, I looked at how the time to solve and memory used increased at the size of the puzzle (N) increased. For each of these tests, an easy puzzle was used (for easy puzzles, I created a valid sudoku grid then removed 25% of the spaces).



*Figure 5: Time vs Puzzle Size*

In this graph, we can see how greedy stayed very constant as the sudoku grid size increased. This is because even though more computations are required at higher sizes, the algorithm failed more as the size and subsequent number of 0 values increased. Divide & conquer increase at the rate that the number of blank spaces increases between grid sizes which aligns with the results seen in Figure 5. The dynamic programming approach increases on the order of O(N4) because the number of computations required by this algorithm scales directly proportional to the size of each row at a rate of N4.

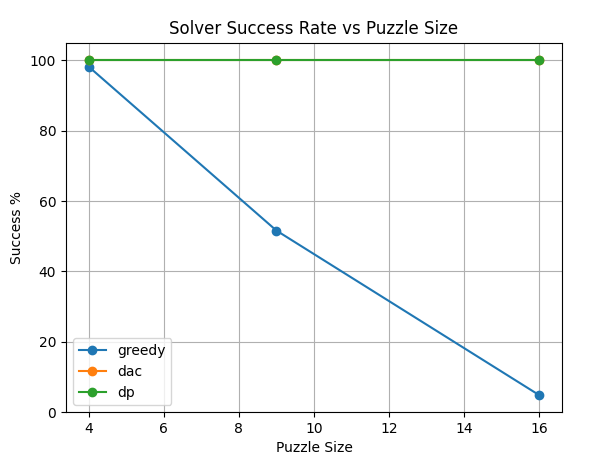


*Figure 6: Memory vs Puzzle Size*

Just like in experiment 1, the divide & conquer and greedy algorithms scale at a rate of O(N2) which is why they are so close together and scale at the same rate. The dynamic programming algorithm requires a lot more memory as more sudoku grid spaces are included because each element in the NxN grid requires up to N elements stored as possibilities in its domain. This results in the peak memory scaling at a rate of O(N3).

**Experiment 3**

In experiment 3, I tested how often each algorithm found a solution. For this experiment, I used a constant easy difficulty. I also tested different N values of 4, 9, and 16. Finally I ran 1000 runs of each experiment and averaged the number of times they resulted in a valid solution.



*Figure 7: Success Rate vs Puzzle Size*

In this experiment, we can see how dynamic programming and divide & conquer approaches find a solution 100% of the time. Greedy looks promising in the beginning with a success rate of 98.2% but it quickly goes down all the way until it reaches a success rate of 4.8% when N=16. Every blank space is assigned some value by the greedy algorithm. With a small domain, the odds of guessing correctly are high which allows the greedy algorithm to correctly solve the sudoku puzzle most of the time when N=4. When the number of possible elements increases, the number of constraining factors also increases. Because this greedy algorithm commits to its ‘greedy’ path without backtracking, it then gets stuck where a cell has ne possible values in its domain.

**Conclusion**

What I learned from this coding assignment was how the greedy and divide & conquer methods of solving this problem performed well at very low sudoku grid sizes. When the grid size is increased by a little, the greedy algorithm quickly loses the ability to find a solution. When increasing the grid size even more or the number of blank spaces, the divide & conquer algorithm will quickly out scale all other algorithms in terms of time needed to complete all computations because the time complexity is O(N!). Although the dynamic programming approach seems to perform worse than both other algorithms in almost all scenarios, these experiments clearly show how dynamic programming becomes far better than both other methods when working with more data. The space complexity still scales at a faster rate, but this doesn’t matter because the greedy approach becomes unusable with low accuracy and the divide & conquer scales too fast to continue working well with large N values.