Sudoku is game where a player has to fill a 9×9 grid so each row, column, and 3×3 box contains 1–9 without repeating.*How to solve sudoku puzzles* (n.d.) The algorithm submitted is to solve a Sudoku puzzle. It uses backtracking search which is a combination of constraint satisfaction and depth-first search (DFS), along with heuristics and certain other optimizations such as pre-calculating neighbours, and others which will be explained. Initially, I started with backtracking, to find possible values for the empty cells. A Minimum remaining values(MRV) heuristic was used to help select the cell with the fewest values remaining at each step, by prioritizing cells that are most constrained, for example, it prioritizes a cell if it has 1 or 2 possible vs. one with 5 possible values.*Constraint Satisfaction* (n.d.)

Initially for easy problems and some medium problems I found success with good solution speeds, but there were still issues with the rest of medium and hard level problems. It was there I realized that some of the problems themselves might be invalid so I added in a validation which checks for duplicates in the puzzle thereby eliminating some of the unnecessary puzzles. A better method for checking possibilities early and a way to reduce backtracking was needed. A least Constraining value(LCV) heuristic along with Forward checking was implemented. LCV allows us to order values for a particular cell by how few neighbours they eliminate, it gives priority to the values that restrict the fewest choices for the neighbouring cells. For example cell (0,3) has domain 2, 4, 6, if 2 appears in 3 neighbouring domains, the score for that is 3, similarly say 4 has score 2 and 6 has score 1, so it orders it as [6,4,2]. This reduces backtracking significantly. Forward checking is another way that significantly contributes to reducing the solution time. Forward checking eliminates conflicting values from all domains of neighbouring cells after we assign a value to a particular cell, basically pruning conflicting values from peers. This makes the MRV more effective. *Constraint Satisfaction Problems* (n.d.)

There were still some more optimizations that were made, apart from validation, this includes pre-calculating the neighboring cells of a particular cell, say for a cell (i, j) we find the row and column neighbors and also the region it is in. This eliminates the repeated calculation of neighbors during the search. I also looked at constraining the domains similar to forward checking, but this is done before backtracking after initializing the domains. This helps remove values in empty cells that can't appear due to numbers existing in their region, or columns or rows, reducing the space and thereby reducing the branching factor when backtracking.

The coursework was quite challenging indeed, it tested my resolve to keep trying to improve that solve time even by the tiniest margin. After researching various heuristics and optimizations that could be made, I believe my code is sound and simple to understand where things are and what they do. The code tackles things from both a logical perspective and also algorithmically. Though further optimizations could be further explored. I came across two further optimizations that could be implemented, changing the standard data structure of using dicts to heap based priority queue (Heapq)*Heapq Algorithm* (n.d.) for the MRV and the other is using bit masking *Bitmasking* (n.d.). Both these methods allow for further optimization and performance improvements. Though Bit masking would lead to more memory overhead due to python integers being objects internally. Heapq would be beneficial in reducing complexity from O(N) to O(logN) for MRV and around O(NlogN) overall in the worst case. Heapq would be beneficial for larger puzzle sizes. Both these approaches would be useful to explore and optimize further. Comparing the metrics from testing the algorithm on two types of machines one being my own local machine, and the other being GitHub codespaces in a virtual environment.

Environment (with specifications)	**Compute Time (sec)**
Local (16Gb Ram, i7 10th Gen, Thinkpad, Fedora Linux)	Approx. 0.9
Codespaces (8Gb Ram, 2 cores, Virtual environment, Ubuntu)	Approx. 1.4

Table 1: Performance metrics on different systems

References

Bitmasking, n.d. [Online]. Available from: https://www.askpython.com/python/examples/python-bit-manipulation-masking-techniques [Accessed 2025-03-27].

Constraint satisfaction, n.d. [Online]. Available from: https://www.cs.cornell.edu/courses/cs4700/2011fa/lectures/05_CSP.pdf [Accessed 2025-03-27].

Constraint satisfaction problems, n.d. [Online]. Available from: https://www.cs.cmu.edu/~15281-s23/coursenotes/constraints/index.html [Accessed 2025-03-27].

Heapq algorithm, n.d. [Online]. Available from: https://docs.python.org/3/library/heapq.html [Accessed 2025-03-27].

How to solve sudoku puzzles, n.d. [Online]. Available from: https://www.sudokudragon.com/sudoku.htm [Accessed 2025-03-27].