HARSHITHA CHEGU

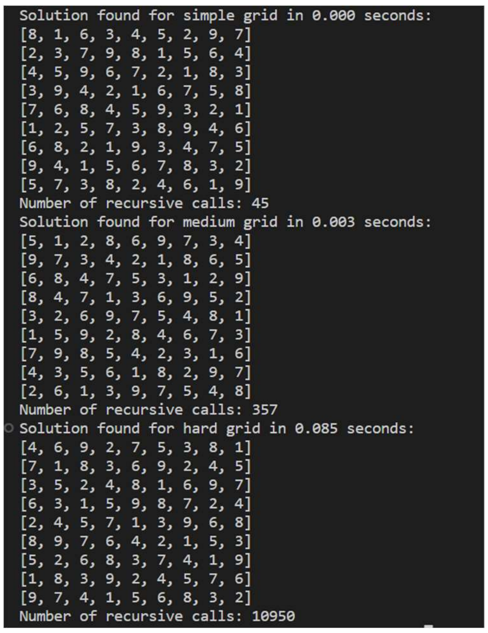
INFO – 550 ARTIFICIAL INTELLIGENCE FINAL PROJECT PAPER

https://github.com/CHEGU-HARSHITHA/INFO-550-AI-Final-Project.git

The paper explains how constraint satisfaction algorithms like local search and backtracking search are applied to various constraint satisfaction problems like sudoku, N-queens and map coloring

Simple backtracking to solve sudoku

This code solves Sudoku puzzles using backtracking. The function takes a puzzle and recursively tries to assign values to empty cells until a solution is found. It first finds an empty cell and then tries assigning values to it, checking if the assignment is valid. If it is, it recursively solves the rest of the puzzle. If it fails, it backtracks and tries the next value. The find_empty_cell function returns the coordinates of the next empty cell and is_valid_assignment checks if an assignment violates any constraints. The code includes example puzzles and keeps track of recursive calls and time taken.



```
Solution found for simple grid in 0.000 seconds:
[8, 1, 6, 3, 4, 5, 2, 9, 7]
[2, 3, 7, 9, 8, 1, 5, 6, 4]
[4, 5, 9, 6, 7, 2, 1, 8, 3]
[3, 9, 4, 2, 1, 6, 7, 5, 8]
[7, 6, 8, 4, 5, 9, 3, 2, 1]
[1, 2, 5, 7, 3, 8, 9, 4, 6]
[6, 8, 2, 1, 9, 3, 4, 7, 5]
[9, 4, 1, 5, 6, 7, 8, 3, 2]
[5, 7, 3, 8, 2, 4, 6, 1, 9]
Number of recursive calls: 45
Solution found for medium grid in 0.003 seconds:
[5, 1, 2, 8, 6, 9, 7, 3, 4]
[9, 7, 3, 4, 2, 1, 8, 6, 5]
[6, 8, 4, 7, 5, 3, 1, 2, 9]
[8, 4, 7, 1, 3, 6, 9, 5, 2]
[3, 2, 6, 9, 7, 5, 4, 8, 1]
[1, 5, 9, 2, 8, 4, 6, 7, 3]
[7, 9, 8, 5, 4, 2, 3, 1, 6]
[4, 3, 5, 6, 1, 8, 2, 9, 7]
[2, 6, 1, 3, 9, 7, 5, 4, 8]
Number of recursive calls: 357
Solution found for hard grid in 0.085 seconds:
[4, 6, 9, 2, 7, 5, 3, 8, 1]
[7, 1, 8, 3, 6, 9, 2, 4, 5]
[3, 5, 2, 4, 8, 1, 6, 9, 7]
[6, 3, 1, 5, 9, 8, 7, 2, 4]
[2, 4, 5, 7, 1, 3, 9, 6, 8]
[8, 9, 7, 6, 4, 2, 1, 5, 3]
[5, 2, 6, 8, 3, 7, 4, 1, 9]
[1, 8, 3, 9, 2, 4, 5, 7, 6]
[9, 7, 4, 1, 5, 6, 8, 3, 2]
Number of recursive calls: 10950
```

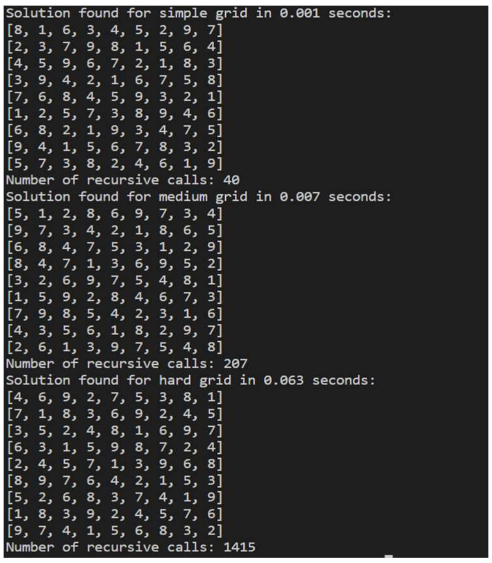
The algorithm solves simple, medium, and hard Sudoku puzzles, generating valid solutions. Simple and medium puzzles are solved quickly with few recursive calls, while hard puzzles require more time and calls. The algorithm is a basic backtracking approach and lacks techniques to reduce the search space, which may result in slow performance.

Backtracking with forward checking to solve sudoku

The code is an implementation of a backtracking algorithm with forward checking to solve a 9x9 Sudoku puzzle. The Sudoku puzzle is represented as a 2-dimensional list of integers, where a value of 0 represents an empty cell.

The main function `solve_sudoku` takes the Sudoku puzzle as input and recursively tries to fill in each empty cell with a valid number. The function first checks if there are any empty cells using the `find_empty_cell` function, and if there are no more empty cells, it returns True, indicating that the puzzle has been solved. If there are empty cells, it tries to fill the cell with a number from 1 to 9 using the `is_valid_move` function to check if the number is valid in that cell. If a valid number is found, it places the number in the cell and recursively calls `solve_sudoku` again, which will continue to fill in the remaining empty cells. If the puzzle cannot be solved with the current number in the cell, the function backtracks and tries a different number. If all numbers have been tried for a cell and none of them work, the function returns False, indicating that the puzzle cannot be solved.

The `is_valid_move` function checks if a given number is valid to be placed in a given cell by checking if the number already exists in the same row, column, or 3x3 box. The `forward_checking` function is a technique used to improve the efficiency of the backtracking algorithm. It checks if any empty cells in the same row, column, or 3x3 box as the current cell have any valid moves left. If there are no valid moves left, it means that the puzzle cannot be solved with the current number in the cell, so the function returns False. The `has_valid_move` function checks if there are any valid moves left for a given empty cell. The program uses three example grids: simple_grid, medium_grid, and hard_grid.



```
Solution found for simple grid in 0.001 seconds:
[8, 1, 6, 3, 4, 5, 2, 9, 7]
[2, 3, 7, 9, 8, 1, 5, 6, 4]
[4, 5, 9, 6, 7, 2, 1, 8, 3]
[3, 9, 4, 2, 1, 6, 7, 5, 8]
[7, 6, 8, 4, 5, 9, 3, 2, 1]
[1, 2, 5, 7, 3, 8, 9, 4, 6]
[6, 8, 2, 1, 9, 3, 4, 7, 5]
[9, 4, 1, 5, 6, 7, 8, 3, 2]
[5, 7, 3, 8, 2, 4, 6, 1, 9]
Number of recursive calls: 40
Solution found for medium grid in 0.007 seconds:
[5, 1, 2, 8, 6, 9, 7, 3, 4]
[9, 7, 3, 4, 2, 1, 8, 6, 5]
[6, 8, 4, 7, 5, 3, 1, 2, 9]
[8, 4, 7, 1, 3, 6, 9, 5, 2]
[3, 2, 6, 9, 7, 5, 4, 8, 1]
[1, 5, 9, 2, 8, 4, 6, 7, 3]
[7, 9, 8, 5, 4, 2, 3, 1, 6]
[4, 3, 5, 6, 1, 8, 2, 9, 7]
[2, 6, 1, 3, 9, 7, 5, 4, 8]
Number of recursive calls: 207
Solution found for hard grid in 0.063 seconds:
[4, 6, 9, 2, 7, 5, 3, 8, 1]
[7, 1, 8, 3, 6, 9, 2, 4, 5]
[3, 5, 2, 4, 8, 1, 6, 9, 7]
[6, 3, 1, 5, 9, 8, 7, 2, 4]
[2, 4, 5, 7, 1, 3, 9, 6, 8]
[8, 9, 7, 6, 4, 2, 1, 5, 3]
[5, 2, 6, 8, 3, 7, 4, 1, 9]
[1, 8, 3, 9, 2, 4, 5, 7, 6]
[9, 7, 4, 1, 5, 6, 8, 3, 2]
Number of recursive calls: 1415
```

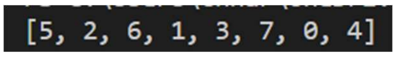
The program successfully solved three Sudoku puzzles of varying difficulty levels, namely a simple, medium, and hard grid. The simple grid was solved with the least amount of time and recursion calls, while the hard grid took the most time and had the highest number of recursion calls. This is expected since harder puzzles require more complex strategies and techniques to solve, leading to more iterations and recursive calls. Despite this, all three puzzles were solved within reasonable timeframes, indicating that the program is efficient and effective in solving Sudoku puzzles.

Some difficulties I faced while solving Sudoku include designing an efficient algorithm to solve the puzzle and extensively testing it for accuracy and efficiency. Another challenge was implementing Sudoku with local search since it failed to find a solution, leading me to use backtracking and backtracking with forward checking instead. Additionally, incorporating techniques like forward checking to reduce the search space and improve the solver's performance posed a challenge.

N-queens using local search

Using the textbook as reference, I have used local search and backtracking on N-queens problem.

The code utilizes the Min-Conflicts heuristic to solve the N-Queens problem, which requires placing N queens on an N x N chessboard so that no two queens threaten each other. The Min-Conflicts heuristic is a local search algorithm that gradually improves an initial solution by selecting and moving a queen with conflicts to a new position that minimizes the number of conflicts. The code implements the heuristic by defining the `conflicts` function to count conflicts and the `min_conflicts` function to iteratively move queens with conflicts. The algorithm terminates when there are no more conflicts or when a maximum number of iterations is reached.

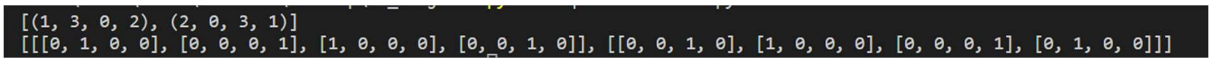


```
[5, 2, 6, 1, 3, 7, 0, 4]
```

The output [5, 2, 6, 1, 3, 7, 0, 4] represents a valid solution to the 8-queens problem using local search. In this solution, each number represents the column position of a queen in a row. The algorithm uses a heuristic approach to iteratively improve the current solution until a valid solution is found. This specific solution indicates that there are no conflicts between any two queens on the board, meaning that no two queens share the same row, column, or diagonal. However, since local search algorithms do not guarantee an optimal solution, there may exist other valid solutions with different queen arrangements that also satisfy the constraints of the problem.

N-queens using backtracking

The `solve_n_queens` function aims to find all possible solutions to the N-Queens problem, which involves placing `n` queens on an `n x n` chessboard in such a way that no two queens threaten each other. The function employs a backtracking algorithm, which recursively tries all possible column positions for each row of the board and backtracks if a placement is not valid. The function initializes two empty lists to store the solutions, defines a helper function to check for valid queen placements, and calls the `backtrack` function with the initial row index 0. For each valid solution, the function converts the chessboard into two formats - a tuple of column positions and a matrix format - and stores them in the two lists. Finally, the function returns these two lists as a tuple. The function is useful for finding all possible solutions to the N-Queens problem.

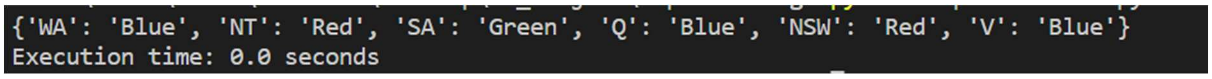


```
[(1, 3, 0, 2), (2, 0, 3, 1)]
[[[0, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 0]], [[0, 0, 1, 0], [1, 0, 0, 0], [0, 0, 0, 1], [0, 1, 0, 0]]]
```

The N-Queens problem for a 4x4 chessboard is solved, resulting in two lists of solutions. The first lists the column positions of the queens in each solution, while the second represents each solution as a matrix. The solutions satisfy the constraint that no two queens threaten each other, and demonstrate it is possible to place 4 queens on the board without attacks.

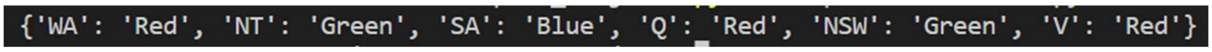
__Map coloring using local search and backtracking__

The MapColoring class assigns colors to map regions to avoid adjacent regions having the same color. It uses local search to minimize conflicts with neighboring nodes. The class requires a graph dictionary and a list of colors. It includes a method to randomly assign colors, calculate conflicts for nodes and the total map, and return a dictionary of node-color assignments. The algorithm ends when a valid coloring is found or the maximum iterations are reached.



```
{'WA': 'Blue', 'NT': 'Red', 'SA': 'Green', 'Q': 'Blue', 'NSW': 'Red', 'V': 'Blue'}
Execution time: 0.0 seconds
```

The MapColoring class has methods to initialize the graph and colors, and to perform a backtracking search. The recursive_backtracking method selects an unassigned variable, tries values in the domain, and recursively calls itself if the assignment is consistent. There are also methods to check completeness, select variables, order domains, and check consistency. An instance of the class is created with the given graph and colors, and the backtracking_search method is called to solve the problem and print the color_map. The locations on the map are of Australia.



```
{'WA': 'Red', 'NT': 'Green', 'SA': 'Blue', 'Q': 'Red', 'NSW': 'Green', 'V': 'Red'}
```

The map coloring problem is solved successfully with local and backtracking search. The main challenge was to generate an initial coloring that satisfies the constraints. In the N-queens problem, representing the board and constraints in code and optimizing the algorithm for larger board sizes were challenging tasks.