Final Project: Sudoku Solver

By: Anmol Tripathi

Git hub link for Project:

https://github.com/unit-mole/Constraint-Satisfaction-Algorithms-using-AI.git

Introduction:

The objective of this project is to implement and compare different algorithms for solving Sudoku puzzles. Sudoku is a popular puzzle game in which the objective is to fill a 9 x 9 grid with digits so that each column, each row, and each of the nine 3 x 3 subgrids contains all the digits from 1 to 9. The algorithms implemented for this project include backtracking, breadth-first search, constraint propagation, depth-first search, forward search, genetic algorithms, local search, and simulated annealing algorithm.

The backtracking algorithm is a brute-force search algorithm that tries to fill the Sudoku grid cell by cell, backtracking when it encounters a conflict. The breadth-first search algorithm uses a queue data structure to explore all possible paths in a level-wise manner. The constraint propagation algorithm reduces the search space by propagating the constraints imposed by the Sudoku rules. The depth-first search algorithm explores the search space depth-wise, using a stack data structure. The forward search algorithm uses heuristics to fill the most constrained cells first. The genetic algorithm uses a population of candidate solutions and applies crossover and mutation operations to generate new candidate solutions. The local search algorithm

iteratively improves a candidate solution by searching for the best neighbouring solution. The simulated annealing algorithm is a probabilistic algorithm that gradually reduces the temperature to escape local optima.

In this report, we will describe the implementation of each algorithm, compare their performance in terms of the number of puzzles solved and the time taken, and discuss the advantages and disadvantages of each algorithm.

Constraint satisfaction Algorithms:

Constraint satisfaction algorithms are a class of algorithms used to find solutions to problems that involve constraints on the values of variables. In general, these algorithms work by iteratively assigning values to variables while checking if the constraints imposed on the variables are satisfied.

The most common examples of constraint satisfaction problems include Sudoku puzzles, scheduling problems, and resource allocation problems. These problems can be modelled as a set of variables, each with a domain of possible values, and a set of constraints that limit the possible values of the variables.

Constraint satisfaction algorithms can be broadly classified into two categories: complete algorithms and incomplete algorithms. Complete algorithms guarantee that a solution will be found if one exists, while incomplete algorithms can only find a solution if one exists within a certain search space.

Some popular constraint satisfaction algorithms include backtracking, forward checking, arc consistency, and genetic algorithms. These algorithms can be used to solve a wide range of real-world problems in fields such as artificial intelligence, operations research, and computer science.

Types of Constraint satisfaction algorithms used for this project:

- Backtracking Search
- Breadth First Search
- Depth First Search
- Forward Search
- Local search

Process used to solve the Sudoku Solver:

1) Backtracking Search:

Below are the steps which I have done to get the results:

The ‘is_valid_assignment()’ function is used to check whether a given number can be assigned to a given cell in the puzzle. It first checks whether the number is already present in the same row or column. Then it checks whether the number is already

present in the same 3x3 sub grid. If the number is not present in any of these locations, it is a valid assignment.

The ‘find_empty_cell()’ function is used to find the next empty cell in the puzzle. It iterates through each cell in the puzzle and returns the row and column of the first cell it finds with a value of 0.

The ‘sudoku_backtracking()’ function is the recursive backtracking algorithm that solves the puzzle. It first finds the next empty cell in the puzzle using the ‘find_empty_cell()’ function. If there is no empty cell, the puzzle is solved and the function returns the completed puzzle. If there is an empty cell, the function tries all possible numbers for that cell. For each possible number, it calls itself recursively with the new number assigned to the cell. If the puzzle can be solved with this assignment, the function returns the completed puzzle. If the puzzle cannot be solved with this assignment, it backtracks and tries a different number. If no number can be assigned to the cell, the function backtracks further to a previous cell and tries a different number there.

The ‘solve_sudoku()’ function is a wrapper function that simply calls ‘sudoku_backtracking()’ and returns the completed puzzle.

Finally, in the main function, a sample Sudoku puzzle is defined and solved using the ‘solve_sudoku()’ function.

2) Breadth First Search:

Below are the steps which I have done to get the results:

The first step is to import the deque class from the collection's module. The deque class is used to create a queue data structure, which is used to keep track of the different possible states of the Sudoku puzzle as the search progresses.

The code defines two helper functions. The first function, "is_valid(puzzle, row, col, num)", checks whether a given number is valid in the specified row, column, and 3x3 sub grid of the puzzle. The function returns True if the number is valid, and False otherwise.

The second function, "next_unassigned(puzzle)", finds the next unassigned cell in the puzzle, and returns its row and column. If no unassigned cells are found, the function returns (-1, -1).

The "bfs_sudoku(puzzle)" function takes the initial Sudoku puzzle as input, and performs a breadth-first search on the puzzle space to find a solution. The function first initializes a queue with the initial puzzle state.

The search continues until the queue is empty. In each iteration, the next puzzle state is removed from the queue using the "popleft()" method. The function then calls the "next_unassigned(puzzle)" function to find the next unassigned cell in the current puzzle. If no unassigned cells are found, the puzzle is solved, and the current puzzle state is returned.

If an unassigned cell is found, the function tries each possible number in the current cell, using the “is_valid(puzzle, row, col, num)” function to check whether the number is valid. If the number is valid, the function creates a new puzzle state with the number in the current cell, and adds the new puzzle state to the queue for further searching.

If no solution is found, the function returns None.

Finally, the code defines the initial Sudoku puzzle, and calls the “bfs_sudoku(puzzle)” function to solve the puzzle. If a solution is found, the function prints the solution. Otherwise, the function prints "No solution found."

3) Depth First Search:

Below are the steps which I have done to get the results:

The first function “is_valid” takes the puzzle board, the row, column and a number as input and returns True if the given number can be placed at the given position without breaking the Sudoku rules, otherwise it returns False. The function first checks if the number is already in the same row or column. If it is, then it returns False, otherwise it checks if the number is already in the same 3x3 sub-grid. If it is, then it returns False. If the number is not in the same row, column or sub-grid, then it returns True.

The second function “solve_sudoku” takes the puzzle board as input and returns True if the puzzle is solved, otherwise it returns False. The function first finds the first unassigned position (i.e. position with value 0) in the puzzle. If all positions are

assigned, then the puzzle is solved and the function returns True. If there is an unassigned position, then the function tries each number from 1 to 9 in that position. For each number, it calls the “is_valid” function to check if the number can be placed at that position without breaking the rules. If the number is valid, then it assigns the number to the position and calls the “solve_sudoku” function recursively to continue solving the puzzle. If the “solve_sudoku” function returns True, then the puzzle is solved and the function returns True. If the “solve_sudoku” function returns False, then the number assigned to the current position is reset to 0 and the function tries the next number. If none of the numbers from 1 to 9 are valid, then the function backtracks to the previous unassigned position and tries a different number. If all numbers have been tried for all unassigned positions and no solution is found, then the function returns False.

In the main function, an example Sudoku board is defined as a list of lists with 9 rows and 9 columns. The “solve_sudoku” function is called with the puzzle as input. If a solution is found, then the solved puzzle is printed row by row. If no solution is found, then the message "No solution found." is printed.

4) Forward Search:

Below are the steps which I have done to get the results:

The function first defines a nested function named ‘is_valid' that checks if a given number is valid to be placed in a cell. This function checks if the number is already present in the same row, column, or 3x3 box.

The function then defines another nested function named 'update_domain' that updates the domain of the unassigned cells. This function updates the domain of the cells in the same row, column, or 3x3 box based on the value that is assigned to a cell.

The function then defines another nested function named 'solve' that implements the backtracking algorithm with forward checking. This function first finds an unassigned cell in the Sudoku grid using the 'find_unassigned' function. If there are no unassigned cells, the Sudoku is solved and the function returns True. Otherwise, the function tries out the possible values for the unassigned cell and checks if the value is valid using the 'is_valid' function. If the value is valid, the function assigns the value to the cell and updates the domain of the neighbouring cells using the 'update_domain' function. The function then recursively calls itself to solve the remaining Sudoku. If the recursive call returns True, the Sudoku is solved and the function returns True. Otherwise, the function backtracks by resetting the value of the cell and updating the domain of the neighbouring cells. If all possible values for the unassigned cell have been tried and none of them lead to a solution, the function returns False.

The function then defines another nested function named 'find_unassigned' that finds an unassigned cell in the Sudoku grid.

The function then initializes the domain of all the cells to True and updates the domain of the assigned cells using the 'update_domain' function.

Finally, the function solves the Sudoku using the backtracking algorithm with forward checking by calling the 'solve' function. If the Sudoku is solved, the function returns the solved grid. Otherwise, the function returns None.

The code then creates an example Sudoku grid and calls the 'forward_checking' function to solve the puzzle. If a solution is found, the function prints the solved grid. Otherwise, the function prints "No solution found".

5) Local search

Below are the steps which I have done to get the results:

The “create_initial_solution(puzzle”) function creates a random initial solution to the puzzle by replacing 0 values in the puzzle with random integers. The “calculate_conflicts(solution, row, col)” function calculates the number of conflicts (i.e., duplicate numbers in the same row, column or 3x3 sub-grid) in the given row and column of a solution. The “calculate_total_conflicts(solution)” function calculates the total number of conflicts in a solution. The “get_best_neighbor(solution)” function finds the best neighbour solution by changing one value in the current solution. The

“solve_sudoku(puzzle)” function solves the Sudoku puzzle using Local Search algorithm by repeatedly finding the best neighbour solution until there are no conflicts left in the solution.

The “if __name__ == "__main__"”: block contains an example Sudoku board as a 2D list, and calls the “solve_sudoku(puzzle)” function to solve the puzzle. Finally, the output is printed as a 2D list, where each row represents a row in the solved Sudoku puzzle.

Results:

After applying various techniques, we were able to get the results from all the algorithms that I have made use of.

Initial Input:

```
[5, 3, 0, 0, 7, 0, 0, 0, 0],
[6, 0, 0, 1, 9, 5, 0, 0, 0],
[0, 9, 8, 0, 0, 0, 0, 6, 0],
[8, 0, 0, 0, 6, 0, 0, 0, 3],
[4, 0, 0, 8, 0, 3, 0, 0, 1],
[7, 0, 0, 0, 2, 0, 0, 0, 6],
[0, 6, 0, 0, 0, 0, 2, 8, 0],
[0, 0, 0, 4, 1, 9, 0, 0, 5],
[0, 0, 0, 0, 8, 0, 0, 7, 9]
```

Output for Backtracking:

```
Time taken: 0.238846 seconds
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]
```

Output for Breadth First Search:

```
Time taken: 0.269512 seconds
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]
```

Output for Depth First Search:

```
Time taken: 0.241665 seconds
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]
```

Output for Forward Search:

```
Time taken: 0.11704707145690918 seconds

[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]
```

Output for Local search:

```
Time taken to solve the puzzle: 0.0 seconds
[5, 3, 4, 4, 7, 5, 4, 7, 7]
[6, 9, 5, 1, 9, 5, 6, 2, 1]
[7, 9, 8, 3, 1, 5, 2, 6, 6]
[8, 6, 7, 2, 6, 6, 7, 6, 3]
[4, 6, 7, 8, 6, 3, 7, 8, 1]
[7, 2, 7, 3, 2, 3, 1, 2, 6]
[3, 6, 7, 7, 9, 6, 2, 8, 7]
[7, 3, 3, 4, 1, 9, 2, 6, 5]
[7, 4, 7, 9, 8, 9, 9, 7, 9]
```

Difficulty in Proceeding with the coding part:

Constraint propagation algorithms, such as AC-3, use the constraints of the problem to reduce the domain of values that can be assigned to each variable. However, in some cases, the constraints may not be sufficient to uniquely determine the values of all variables, leading to multiple possible solutions or even no solutions.

Constraint propagation algorithms have high time complexity, especially when dealing with complex constraints or large domains. As a result, the algorithm takes a long time to converge or may not be able to converge at all within a reasonable time frame, making it unsuitable for practical use.

Genetic algorithms rely on random mutations and crossovers to generate new solutions, but in a Sudoku puzzle, each cell has to have a specific value that must satisfy the constraints of the puzzle. Additionally, the number of possible solutions for a Sudoku puzzle is very high, making it difficult for genetic algorithms to explore the search space efficiently. Therefore, genetic algorithms can be used to solve Sudoku puzzles, but they may not always be the most efficient or effective method compared to other algorithms like backtracking.

Conclusion:

Based on the results of the Sudoku solver, we can see that Backtracking, Depth First Search, and Breadth First Search algorithms are able to solve the Sudoku puzzle correctly in a reasonable amount of time. However, Forward Search algorithm seems to be significantly faster than other algorithms as it exploits the constraints of the Sudoku puzzle to eliminate large portions of the search space.

On the other hand, the Local Search algorithm does not seem to be an appropriate algorithm for solving the Sudoku puzzle as it is unable to solve the puzzle correctly within a reasonable amount of time. This could be due to the fact that Local Search algorithm only focuses on finding the best neighbour solution at each iteration without considering the global solution space, which could lead to a sub-optimal solution.

Therefore, based on these results, it can be concluded that algorithms like Backtracking, Depth First Search, Breadth First Search, or Forward Search are better suited for solving the Sudoku puzzle than Local Search algorithm.