Sudoku solving AI agents

Written by Maheshwari Panchal

[1]Northeastern University
panchal.ma@northeastern.edu

Abstract

This project explores the application of Artificial Intelligence (AI) agents to solve Sudoku puzzles using different algorithms. The project focuses on three main algorithms: backtracking, constraint satisfaction, and Knuth's Algorithm X. It analyzes their strengths, limitations, and computational complexities, examining how each algorithm performs. The findings will contribute to understanding the trade-offs and performance characteristics of diverse AI techniques when applied to similar constraint-satisfaction problems.

Introduction

The Sudoku-solving agent is an example of how artificial intelligence and search algorithms can be applied to solve complex puzzles efficiently. Its results will demonstrate its effectiveness in solving Sudoku puzzles of varying difficulty levels. The best example of AI is to test and decide the best move to perform a task. Hence I wanted to use AI to solve the famous Sudoku puzzle.

The task environments of the "Sudoku playing agent" can be characterized as:

- Fully observable
- Deterministic
- Sequential
- Static
- Discrete
- Single-agent
- Known Environment

Sudoku is a popular logic-based puzzle in which a 9x9 grid must be filled with digits from 1 to 9 such that each row, each column, and each 3x3 subgrid (region) contains all digits without repetition. The computational perspective of this problem is to find a valid solution that adheres to Sudoku rules.
The Sudoku puzzle can be defined as:
State space: The state space consists of all possible configurations of a 9x9 Sudoku grid. Each cell in the grid can contain a digit from 1 to 9 or be empty. The state space is constrained by the Sudoku rules, ensuring no digit repetition in rows, columns, or 3x3 regions.
Initial State: The initial state is an unsolved Sudoku puzzle with some cells pre-filled with digits (initial clues). These initial clues represent the starting configuration of the Sudoku grid.
Actions: The agent's actions consist of selecting an empty cell in the grid and assigning a digit from 1 to 9 to that cell. The agent may also choose to clear a cell by setting it to an empty state. These actions represent the agent's decisions during the puzzle-solving process.
Goal State: The goal state is a fully filled 9x9 Sudoku grid that adheres to the Sudoku rules. In the goal state, each row, each column, and each 3x3 region contains all digits from 1 to 9 without repetition. The agent's objective is to reach this goal state.
Constraints: The agent must follow the Sudoku constraints, which include:

- No digit repetition in rows.
- No digit repetition in columns.
- No digit repetition in 3x3 regions.

The agent cannot violate the initial clues provided in the puzzle.

Sudoku Boards

Sudoku problems are sometimes divided into several difficulty categories according to how intricate they are. A Sudoku board's degree of difficulty is determined by several variables, such as the methods or approaches needed to solve it, the quantity and arrangement of provided hints, and the logical inferences required to solve the puzzle.

- **Easy:** Easy Sudoku puzzles typically have a sizable portion of the grid filled in using provided hints. The problem-solving strategies needed to solve these puzzles are really easy and include basic reasoning and simple elimination. The sudoku board that I used, has 36 initial clues.
- **Medium:** Compared to easy Sudoku puzzles, medium-level Sudoku problems include fewer beginning hints, making them marginally harder. More sophisticated techniques must be used by solvers, such as locating concealed singles, naked pairs or triples, or using more intri-

cate logical inferences. The sudoku board that I used, has 30 initial clues.

- **Difficult:** Difficult Sudoku problems have fewer initial hints than those that are medium and easy. To progress, one needs to use advanced strategies like Swordfish, X-Wing, or other sophisticated methods. A lot of substantial logical reasoning is required to solve them. The sudoku board that I used, has 25 initial clues.

  Along with the above widely used Sudoku problems, I implemented the algorithms on some hardest available Sudoku problems. Some of them are listed below:

- **AI Escargot:** Named by AI researchers as the "world's hardest Sudoku," this puzzle gained fame for its extreme level of difficulty. It's characterized by minimal given clues but can be solved using advanced logic and techniques. It has 22 initial clues.
- **Arto Inkala:** The Finnish mathematician Arto Inkala, gained attention for its challenging nature. It has very few initial clues but is solvable through logic. It's known for being one of the toughest Sudoku puzzles. I used two versions of the Arto Inkala problems with 21 and 22 initial clues
- **Steering Wheel:** This is also one of the hardest Sudokus. It has 19 initial clues.
- **Blonde Platine:** This puzzle has 21 initial clues and is also one of the hardest Sudokus.

The code for the above Sudoku puzzles can be found in the python file sudokuboards.py. This file is available in the link provided in the Supplementary Materials section.

Comparison of the puzzles and agents

- Solving techniques: The number of basic and advanced solving techniques required to solve the puzzle influences its difficulty.
- Solution Path: The number of steps and the sequence of logical deductions needed to reach the solution can indicate difficulty. Puzzles that require a longer sequence of steps or combinations of different strategies may be considered more challenging.
- Human solvability: Sudoku puzzles are often graded by how easily they can be solved by a human solver without guessing or applying brute force. Puzzles that require creative thinking and combinations of various techniques without resorting to trial-and-error methods are generally considered more difficult.
- Generation Algorithm: Assign difficulty based on the algorithms, puzzles and structures used to generate the puzzle
- Solving Time: Difficult problems take more time to solve

In this project, I've compared the performance of the different algorithms on the Sudoku puzzles based on time required to solve them.

```
Algorithm 1: Backtracking
Input: 9x9 Sudoku grid with initial clues
Output: Solved Sudoku OR "No solution for the given Sudoku"
 1: if Grid is complete then
 2:     return Solved Sudoku
 3: end if
 4: Select a Valid move
 5: for Fill in a number from 1 to 9 do
 6:     if Number abides by the constraints then
 7:         Fill in the number at that cell
 8:         if Grid is complete then
 9:             return Solved Sudoku
10:         end if
11:         return 0 and Backtrack to the last correct value and
            update cell
12:     end if
13: end for
14: return "No solution for the given Sudoku"
```

Agents

Various types of algorithms or agents can be employed to solve Sudoku. Some well-known algorithms for solving Sudoku are: Depth First Search, Brute force, Backtracking, Parallel Processing, Human-like solving, Constraint Propagation, Algorithm X and Heuristic Search. In this project, I've implemented three algorithms, namely Backtracking, Algorithm X, and Constraint Propagation on the different difficulty levels of Sudoku problems listed above. These algorithms are discussed in more detail below.

1. Backtracking

Brute force is checking every possible solution till you find the right one. Backtracking is brute force with clever optimization. In Backtracking, we don't generate all possible solutions, rather than we keep go on checking whether the solution is correct or not at every step. If it is correct, we continue generating subsequent solutions to the puzzle. If it is incorrect we backtrack to the previous step and check for the other solutions. This prevents generating invalid recursion subtrees and saves a lot of time. A recursive algorithm that systematically explores potential solutions. The pseudocode of the algorithm is provided in 'Algorithm 1'.

Results: On implementing the different Sudoku puzzles on this algorithm, I found that almost all solutions were available in a few milliseconds. More details on the time required to solve each of the above puzzles are listed in Table 1, after the Agents section.

The only puzzle that could not be solved using this algorithm was the Steering Wheel. Due to the countermeasure added, a timer was added to add an upper limit on the solving time. This helped in stopping the code when the Steering Wheel board could not provide a result after 18secs.

2. Knuth's Algorithm X

This algorithm was created by Prof. Donald Knuth and is widely known as the Dancing Links Algorithm. It gets this

Algorithm 2: Dancing Links

Input: 9x9 Sudoku grid with initial clues
Output: Solved Sudoku OR "No solution for the given Sudoku"

```
1:  if No Solution exists then
2:      return "No solution for the given Sudoku"
3:  else
4:      return Solved Sudoku
5:  end if
6:  Convert Sudoku to exact cover problem
    Each cell is doubly linked to neighboring cells
    Each row represents a constraint, each column represents a cell-value pair
    Populate the grid based on Sudoku constraints
7:  if Grid is complete then
8:      return Solved Sudoku
9:  end if
10: Cover the column
11: for each row in column do
12:     Remove row
13:     if Grid is complete then
14:         return Solved Sudoku
15:     end if
16:     Uncover the row and column
17: end for
18: return "No solution for the given Sudoku"
```

name due to its doubly linked data structure and while the algorithm is in execution, the doubly linked cells, cover and uncover the specific rows or columns, based on the defined constraints, in such a way that it seems that the cells are dancing. The sudoku problem can also be defined as the exact cover problem. It is a specific type of constraint satisfaction problem where all constraints have to be met and no constraint can be met more than once. The exact cover problem is a combinatorial problem in mathematics and computer science that involves finding a subset of a given collection of sets such that each element in the universal set is covered exactly once.

Here, the goal is to cover all cells (or constraints) with values (or elements) satisfying certain rules (or conditions). It employs dancing links which facilitates backtracking efficiently by linking rows and columns in a specific way. In order to identify solutions that fulfill every constraint (column), it methodically investigates every conceivable combination of elements (rows). Algorithm X's (with dancing connections) primary benefit is how well it manages backtracking and solution space exploration. By methodically choosing rows and columns to cover limitations and reverting when inconsistencies arise, it effectively reduces the size of the search space. The pseudo-code of the algorithm is provided in 'Algorithm 2'.

Results: On implementing the different Sudoku puzzles on this algorithm, I found that all solutions were available in a few milliseconds. More details on the time required to solve each of the above puzzles are listed in Table 1, after the Agents section. These results were in line with the expected results. Hence the Dancing links algorithm can now successfully be implemented on larger sudoku puzzles to judge its efficiency

Algorithm 3: Constraint Propagation

Input: 9x9 Sudoku grid with initial clues
Output: Solved Sudoku OR "No solution for the given Sudoku"

```
1:  if Grid is not valid then
2:      return Inconsistent Sudoku
3:  else if Grid is complete then
4:      return Solved Sudoku
5:  end if
6:  if Value is valid based on Sudoku constraints then
7:      Populate the cell with that value
8:  else
9:      return "No solution for the given Sudoku"
10: end ifAssign possible values to all empty cells
    Implement heuristics to get final values faster
11: for Iterate through all the cells do
12:     Select the cell with Minimum Remaining Value
13:     if MRV>1 then
14:         Sort domain values by Degree Heuristic
15:     end if
16: end for
17: return "No solution for the given Sudoku"
```

3. Constraint Propagation

This algorithm works as follows: first, the grid is initialized. This includes testing if the provided grid is valid or not. If it is valid, the empty cells are populating with all the possible values. This is done using simple elimination. Here we iterate through the grid and apply constraint propagation to reduce the possibilities for each cell. This is done by eliminating values that are not valid based on the initial clues and existing placements. For each cell with a single value (given in the initial clues), remove that value from the possibilities of its peers (cells in the same row, column, or region). Repeat this process until no further changes can be made through constraint propagation.

If the Sudoku puzzle is not completely solved after constraint propagation, start a search using backtracking to find a solution. In this project, I've implemented depth first search along with backtracking to solve the resulting grid. Whenever a contradiction is reached (e.g., a violation of Sudoku rules), we perform backtracking to the previous state and try a different value in the current cell. This process is continued recursively exploring the search space until a solution is found or all possibilities are exhausted.

It uses a depth-first search that is capable of exploring and evaluating assignments, which can now be thought of as nodes. In each iteration, the Solver checks that the assignment is consistent, i.e. that it adheres to the constraints. If the current assignment is not consistent (does not adhere to the constraints), the algorithm backtracks to the last known consistent state and tries a different path. The pseudo-code of the algorithm is provided in 'Algorithm 3'.

After implementing this algorithm on the boards, I found that they had a similar results to the Dancing Links algorithm. So, to improve these results, I added a few heuristics to improve the performance.

Heuristics The Solver utilizes two heuristics for variable ordering, i.e. choosing which empty position to fill next. These are the minimum-remaining-values (MRV) and degree heuristics. By adding these heuristics and replacing the previous static variable ordering, the runtime of the solution improved greatly.

- MRV: The minimum-remaining-values (MRV) heuristic, also called the ”most constrained variable” heuristic, picks the variable with the fewest possible values. [2] In doing so, it picks the variable that has the highest likelihood of failing soon, reducing the number of computations otherwise spent searching other variables first. By extension, if a variable has no possible values left, then it would be selected and fail straight away. Note that multiple variables can have the same number of possible values, and as such the MRV heuristic may return a list of variables. In this case, the degree heuristic is used as a tie-breaker.
- Degree: The degree heuristic selects the variable with the most constraints on other unassigned variables, i.e. the one that affects the greatest number of empty positions.

Results: On implementing the different Sudoku puzzles on this algorithm, I found that all solutions were available in a few milliseconds. More details on the time required to solve each of the above puzzles are listed in Table 1, after the Agents section.

Comparison of the Algorithms

Table 1 provides a comparison between these algorithms in terms of solving time. It is also important to check the efficiency and applicability of these puzzles and algorithms. After a thorough review of the three algorithms, we can say that:

- Backtracking: This algorithm works well for easy to medium problems. As the difficulty increases, the solving time also increases. Also, one of the hardest Sudoku puzzles could not be solved by this algorithm. This is recommended for beginners and puzzles with a low number of empty cells.
- Knuth’s Algorithm X: This algorithm works well for all types of puzzles. This is recommended for puzzles with a moderate number of empty cells.
- Constraint Propagation: Once heuristics were added to this algorithm, a much superior performance was obtained. This algorithm works well for all difficulty levels, including hard puzzles. This is recommended for puzzles with a high number of empty cells.

In Figure 1, the solving time is plotted against the difficulty of the Sudoku problems. The numbers on the y-axis denote the number of initial clues. As this number increases, the difficulty of the Sudoku increases. Hence from the plot we can see that for Backtracking, the time increased as the difficulty

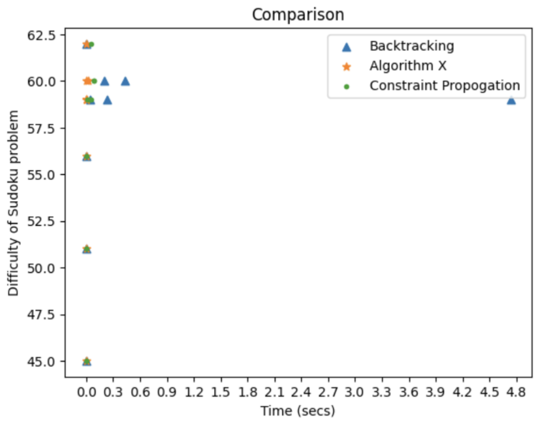

Figure 1: Comparison of the three algorithms on the different Sudokus based on time required to solve them.

.

increased. Whereas the Constraint Propagation and Dancing Links algorithms performed well in spite of the increase in the difficulty.

Conclusion

In this paper, we presented the Python implementation of Backtracking, Algorithm X, and Constraint Propagation in solving a Sudoku puzzle. Through empirical evaluation and comparative analysis of these AI-based approaches, this project aims to provide insights into the strengths and weaknesses of different algorithms for solving Sudoku puzzles. The findings are that Constraint Propagation along with heuristics and backtracking works well for Sudoku and thus, can be extended to other Constraint satisfaction problems. When the Sudoku is converted to an exact cover problem, the Dancing Links algorithm also performs pretty well.

These results are in line with my expectations. I look forward to implementing the Dancing Links algorithm to a 16x16 and 25x25 Sudoku puzzle and verifying its optimality.

Supplementary Materials

Click here to access the code:

- Sudoku Agents: This file contains the three agents or algorithms as classes. The Pseudo code that is provided above, is coded in these classes for the three algorithms
- Sudoku Boards: Contains the lists of all the boards listed in the ”Sudoku Boards” section. These boards are presented in an array and string format
- Sudoku tests: This file accesses the above two files and sequentially solves each board using each agent. It also contains a timer to test the solving time for each of these algorithms.

References

[1] https://www.ocf.berkeley.edu/ jchu/publicportal/sudoku/0011047.pdf

Algorithm	Backtracking	Knuth's Algorithm X	Constraint Propagation
Easy	0.002s	0.004s	0.001s
Medium	0.005s	0.004s	0.001s
Difficult	0.006s	0.003s	0.002s
AI Escargot	0.039s	0.005s	0.005s
Arto Inkala	0.43s	0.018s	0.009s
Steering Wheel	–	0.003s	0.001s
Blonde Platine	4.736s	0.006s	0.003s

Table 1: Time for solving the Sudoku

[2] MichaelEmery.pdf (montana.edu)

[3] GitHub - sg2295/Sudoku-Solver: An agent that can solve Sudoku puzzles, following a backtracking search, using a combination of depth-first search and constraint propagation.

[4] Sudoku Generator - Puzzle Maker - Printable Sudoku Puzzles (printablecreative.com)

[5] sudoku-generator · GitHub Topics · GitHub

[6] sudoku-generator/Sudoku/Board.py at master · RutledgePaulV/sudoku-generator · GitHub

[7] Understanding the basics of Linked List - GeeksforGeeks

[8] python - Algorithm for solving Sudoku - Stack Overflow

[9] What are the criteria for determining the difficulty of Sudoku puzzle? - Puzzling Stack Exchange

[10] sudoku-solver/solver.py at master · iahsanujunda/sudoku-solver · GitHub