INFO 550: AI final project – Building & Comparing Sudoku Solvers

Akshita Sharma

May 6, 2023

GitHub repository

https://github.com/AkshtaSharma/AiFinalProject

Code and paper have all been uploaded to this public repository. Please copy paste this link and don't click on it directly as it won't work.

Introduction

Sudoku is popular combinatorial type challenging puzzle in which players use logical reasoning and strategic planning to fill a 9x9 grid with integers from 1 to 9. The rules of the game are simple: No integer from 1-9 must be repeated in any row, column, or 3x3 grid outlined within the 9x9 grid and all integers must be present in each row, column, and 3x3 grid once. The simplicity in the rules of the game is from where the complexity also arises. This game has been a popular endeavor among players: both who play using pen and paper, and those who play using computational intelligence. In this paper we describe two methods to solve this game algorithmically. We explain and implement a solution that uses back tracking and another that uses back tracking combined with constraint propagation to solve the problem. In the next few pages we provide an overview of the Sudoku game, the fundamentals and the details of the

algorithms we've used to build those solvers, and explain their implementations in Python. Furthermore, we compare the two solvers in terms of performance analysis using their solving time and efficiency.

Solving Sudoku Using Backtracking

Backtracking is a sophisticated technique in artificial intelligence applications wherein the problem is solved efficiently by being broken down into smaller and simpler subproblems [2]. This technique is especially helpful in problems in which the solution space is simply too large for a brute force approach or an exhaustive search strategy. The idea behind backtracking is to iterate through all possible solutions to incrementally build a candidate solution. As soon as a candidate solution is built that compromises the constraints of the problem at any given time, the algorithm reverts back to the last known (partial) solution that satisfied all the constraints and considers a different possible solution to continue its search for a potential solution.

Backtracking is an algorithm that is better defined by its algorithmic structure than it is using some mathematical equation. The general technique that the algorithm follows is a recursive one that involves exploring all possible solutions to a problem by incrementally building a partial solution and undoing incorrect or unsatisfactory choices as necessary. The solution is found when a potential solution is generated that satisfies all the constraints. The algorithm either terminates if that solution is found or terminates if all possible options have been exhausted.

Now we describe how we implement Backtracking to build a Sudoku solver. The general technique of a Backtracking implementation to solve a Sudoku

puzzle involves filling in the empty cells of the puzzle with numbers one by one, starting with the first cell. For each cell, the algorithm chooses a number that doesn't violate any of the constraints, i.e. it ensures that no digit 1-9 is present more than once in any row, column, or 3x3 grid. The process continues until all cells are filled. If, at any point, no number can be placed in a cell without violating any of the constraints, the algorithm back tracks to the last stable game state and tries a different number for the cell changed in that state. Though it is hypothesised that the order in which the empty cells are filled effects the efficiency and effectiveness of the algorithm, that has yet to be explored in a future iteration of this project. This process continues until all cells are filled or until all possibilities have been exhausted.

The code for this project is provided in the GitHub repositiory linked above.

Solving Sudoku Using Backtracking + Constraint Propagation

In this section, we consider another method to algorithmically solve the Sudoku puzzle. In this case, we want to try and solve the problem more efficiently and effectively than just using backtracking, which in a sense is just a modified form of a brute force approach. Using the fact that Sudoku is a game that is propelled using a set of constraints, here we consider employing constraint propagation algorithm [1]. The constraint propagation algorithm essentially reduces the number of possibilities to be explored for each set by building the set of potential numbers for each cell by only including those that satisfy the set of constraints in the game. In the case of Sudoku, the set of possible values for each cell will be determined based on the values already filled in the same

row, column, and subgrid containing the cell under consideration. Doing this narrows down the possible options for the algorithm to explore when solving Sudoku leading to a quicker solution to the problem, i.e. a faster elimination of all the invalid numerical combinations.

To develop this solver, we consider combining backtracking with constrain propagation to create a more efficient solving algorithm. The main idea here is to limit the solution space for each cell by propagating the constraints across the board. The goal of this is to minimise the solution space for each cell so as to optimise the search for the right solution in the backtracking step. Combining backtracking with constrain propagation optimises the solver as it reduces the number of branches to be explored in the search tree when solving the puzzle therefore reducing the computational complexity of the problem. In the next section we provide a detailed account of the computation complexities of both methods and and even compare their performances when Sudoku boards of various difficulty levels.

Comparing the Two Methods

In this section we compare the efficiencies and computational complexities of the brute force method of solving Sudoku with the two methods we have developed and explained above. The computational complexity of solving Sudoku using brute force is NP-complete. The brute force method to solve Sudoku would generate all possible combinations of numbers in the empty cells and test each outcome until a valid solution is found. With the brute force technique, the number of possible combinations increases exponentially with the size of the grid. Therefore, in the worst case scenario the computation complexity would be $O(9^{81})$, which is an astronomical number. This complexity makes it an im-

practical technique to solving large and complex puzzles like this.

To overcome this issue, we first employ backtracking. Backtracking is better than brute force in Sudoku because it significantly reduces the search space by only exploring paths that are likely to lead to a solution. In other words, backtracking algorithm uses constraints to prune the search space, rather than blindly searching through all possible solutions. The computational complexity of backtracking when applied to Sudoku is given as $O(9^m)$ where m is the number of empty cells. In a typical game, only about 30-40 cells are empty which makes the computational complexity of this strategy a lot more manageable.

With the addition of constraint propagation techniques, the number of possibilities for each empty cell is significantly reduced, which further improves the efficiency of the algorithm. In other words, the addition of constraint propagation techniques significantly reduces the search space and improves the efficiency of the backtracking algorithm. While the precise complexity is still given as $O(9^m)$, the efficiency of the backtracking + constraint propagation algorithm combined proves to be numerically better through a multitude of examples as we will see below. In practice, the efficiency of the backtracking algorithm with constraint propagation varies depending on the puzzle's difficulty (number of unknown cells, location of pre-filled cells, etc.).

Below let's look at some results.

Now that we have shared the mathematical formulas precisely describing the computational complexities of these algorithms, we compare the performance of running the backtracking strategy alone with the strategy of running it in

conjunction with constraint propagation using the time (in seconds) it takes the code to run. We don't explicitly run the brute force algorithm to demonstrate that it is slower than the other two approaches as we consider that to be widely known and acknowledged.

Below are the boards that we consider along with the time (in seconds) it took each algorithm to solve the puzzle. We get the times using Python's *time.time*() function.

```
board = [
    [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, 0]]
```

The above puzzle took 0.07421588897705078 seconds using backtracking alone and 0.06482195854187012 seconds using a backtracking + constraint propagation strategy.

In the above example we get the following solved Sudoku board:

```
[[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]]
```

The next puzzle we consider is below.

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

Notice that the above puzzle has a lot more cells with no given value. Therefore, backtracking along would have to iterate through many more possible options before arriving to the right solution. This is a classic example in which applying constraint propagation techniques can quickly narrow down the possible values for each cell, speeding up the solution finding process.

The above puzzle took 765.8333899974823 seconds using backtracking alone and 663.6948480606079 seconds using a backtracking + constraint propagation strategy.

In the above example we get the following solved Sudoku board:

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

The above examples are two of many such Sudoku puzzles that can be constructed to show the efficiencies and advantages introduced to the solution strategies when employing backtracking and/or constraint propagation as opposed to simply using a brute force approach.

References

[1] David Carmel. Solving Sudoku by Heuristic Search — david-carmel. https://medium.com/@davidcarmel/solving-sudoku-by-heuristic-search-b0c2b2c5346e. [Accessed 06-May-2023].

[2] Anil Kemisetti. Solving sudoku ... think constraint satisfaction problem, Mar 2018.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A modern approach*. Pearson, 2021.