Algorithms and Data Structures - Assignment 1
Sudoku Solver

Introduction

At its core, Sudoku presents players with a square grid divided into columns, rows, and smaller sub-grids. Initially, some cells are filled with numbers. The goal is to fill in the empty cells with numbers from 1 to n, n being the size of the grid, ensuring no repetition within any column, row, or sub-grid. While Sudoku is typically played on a 9x9, it can be adapted to any square grid with a side length being a perfect square.

In terms of algorithmic thinking, the starting state can be represented as a 2-dimensional data structure (e.g. 2D array or a list of lists) filled with natural numbers following the uniqueness constraints, or 0's representing empty spaces to be filled in. To take an action towards the solution, it is necessary to iterate over the empty cells and fill them with numbers that do not violate the Sudoku no repetition rule, until the grid is complete (there are no empty (0's), left).

In this assignment, we will explore two common approaches to solving Sudoku:

1. Exhaustive Search - trying every possible number combination, and
2. Backtracking - which initially also tries every possible combination, but abandons paths that do not lead to a valid solution.

We will compare those algorithms and theorise about possible improvements that can be made.

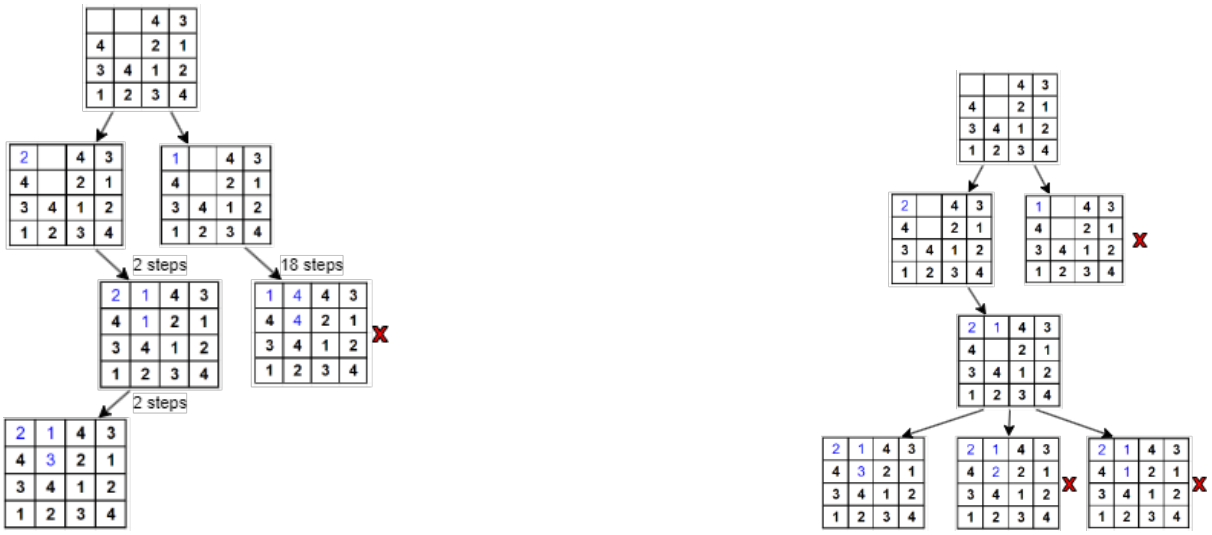
1 On differences between Exhaustive Search and Backtracking

Exhaustive Search, also known as Brute Force, is a problem-solving paradigm, which involves checking every single possibility within the constraints of a system. It is guaranteed to find a solution if one exists, but at the cost of exponential complexity as the size of the problem increases. The approach we have implemented fits the Exhaustive Search criteria as every cell is filled until the grid has no empty spaces left, at which point it checks whether the completely filled grid is valid, returning either *True*, if it is, or *False*, if it does violate Sudoku rules.

Backtracking on the other hand is a more refined approach which involves elimination of paths that do not lead to a valid solution at an earlier stage. It allows not to wait until the end-state is reached to check the solution, as compared to Exhaustive Search.

The implementation details do slightly differ in between those two approaches. It boils down to the point at which the solution is validated. For Exhaustive Search, the validity of the grid is only checked at the very end, once all of the cells have been filled. In backtracking, however, it is done immediately after a new solution to a cell is tried, as well as at the end state. If the partial solution violates the rules, the cell modification is reverted and the path gets discarded (further combinations with this solution are not explored).

To further analyze these two approaches, we have prepared two simplified search trees 1b representing each method. Due to physical constraints of this paper, (Figure A: Exhaustive Search) is severely truncated, while (Figure B: Backtracking) is fully represented.



(a) Exhaustive Search

(b) Backtracking

Figure 1: Search tree diagrams

The Exhaustive Search tree is notably more extensive, as it attempts to fill every cell serially, without considering the viability of a partial solution. This effectively represents the exhaustive nature of the tree, but also results in exponential growth.

The Backtracking tree is seemingly more compact. It prunes the tree, stopping the further exploration of a branch as soon as the partial solution violates the rules. This way, Backtracking allows us to reduce the number of steps to find a valid solution, and shows the optimality of Backtracking for Sudoku particularly. A solution path is pursued only if it constantly continues to satisfy the requirements, proving that this approach is optimal in the sense that it will find a solution if one exists, without doing unnecessary computations to pursue invalid paths.

Backtracking, therefore, is most commonly a better solution in terms of performance and works especially well for problems where the rule violations can be quickly identified, further ensuring easy navigation through the solution space and pruning of branches.

2 Complexity

The complexity of both Exhaustive Search and Backtracking can be described in terms of the Big O notation. In our specific Sudoku problem, with m representing the number of empty squares and n the number of possible numbers to try (the length of a row of the grid), the complexity seems to be $O(n^m)$ for both algorithms. This would suggest that for each of the empty squares, there are n possible solutions to explore, effectively leading to an exponential growth in the search space.

However, in reality, the Backtracking approach allows to avoid exploring branches of the tree that do not lead to a valid solution. This pruning can significantly influence the number of computations, especially with larger problems. The $O(n^m)$ is the worst-case scenario, but the complexity is often much lower, depending on the initial configuration of the Sudoku, in particular the percentage of cells filled. With enough initial cells filled in, backtracking can quickly prune most of the branches, effectively performing closer to linear or polynomial time in the average case [2]. All in all, backtracking will always perform equally to or better than Exhaustive Search, which makes it a more optimal choice for the Sudoku problem.

3 Empty Grid Scenario

State space is a set of all possible unique configurations of a system. In most cases, for an Exhaustive Search algorithms, the state-space size is the same as the worst-case time complexity. In our case that would be n^m with m being the number of empty squares and n being the quantity of possible numbers to try. For a standard 9×9 Sudoku grid that number would 9^{9^2} or 9^{81}, since all of the 81 cells can contain any of the 9 possible numbers. This is an astronomical number that would require an astronomical amount of time to go through all or even a fraction of the possibilities. Thus, realistically speaking it is impossible to get the solution in real-time.

Further Theoretical Improvements For The Algorithm

When it comes to further improvements of our algorithm there are two approaches that are the most promising, both based on further pruning and thus spending less time on dead ends:

- Each cell can be assigned a number of possible values that would not break the rules. Starting the iteration from cell's with the smallest values will dramatically increase the efficiency of such algorithm, by leading to earlier elimination of invalid paths.
- Implementing human solving techniques as constrains. For example "Naked Single"[1] technique can be used. By finding the cells where only one value is applicable, it is possible to find correct values for cells earlier, when compared to guessing.

Those are just simple examples on how the algorithm could be further improved, however there are also multiple other techniques that can be used to further improve the performance with various degrees of implementation difficulty

Summary and Discussion

In conclusion we have reviewed and compared two popular and related algorithmic paradigms - Exhaustive Search and Backtracking - using Sudoku as our benchmark for their performance. We found Exhaustive Search an extremely easy approach to take when it comes to development as it doesn't require much thought, but it is only a viable option for relatively small problems due to its exponential growth in complexity, and thus time required to complete. Backtracking on the other hand, being slightly more complex of the two, provides a sizable boost in efficiency, effectively outperforming Exhaustive Search by avoiding unnecessary computations. We have also theorised further ways to improve the algorithm and further increase the efficiency, by introducing more sophisticated pruning methods and integrating human solving heuristics.

Overall, the Sudoku served as a helpful benchmark for comparing and examining the algorithms. The conclusions made highlight the importance of choosing the right algorithmic strategy for a given problem size, as well us the underlying trade-offs between simplicity and computation efforts.

References

[1] "Naked Single". In: *Sudopedia, the Free Sudoku Reference Guide* (2008).

[2] Peter Norvig Stuart Russell. "Artificial Intelligence: A Modern Approach (4th Edition)." In: (2020).