# CSCE - 2211 Spring 2025 Term Paper
# Sudoku for Backtracking

Dalia Hassan Youssef Aglan Malak Abdelhalim Rana Hamed Menna Essam
Department of Computer Science and Engineering, The American University in Cairo

***Abstract*—This paper examines the application of data structures and algorithms in solving Sudoku puzzles, emphasizing recursive backtracking and tree-based search methods. The literature review addresses well-known solving techniques like constraint propagation, brute-force search, and recursive backtracking, discussing their effectiveness and constraints in the realm of constraint satisfaction problems. The developed system makes use of a recursive backtracking algorithm, enhanced by a backtracking tree that represents the decision-making process as a dynamic recursive structure. Each tree node captures a specific board state, enabling a clearer understanding of the solution pathway.**

**A custom dynamic array class (`Vector`) is designed to manually manage memory, substituting for standard container libraries, thus emphasizing the fundamental implementation of data structures as part of the course focus. The Sudoku board is represented as a two-dimensional grid using nested `Vector` instances, facilitating cell-level operations and input validation. The system includes a command-line interface for puzzle engagement, featuring board visualization, move validation, reset options, automatic solving, and customizable puzzle generation.**

**An experimental comparison investigates the standard backtracking approach against the tree-based variant, assessing both versions in terms of recursive depth, memory usage, and solution process traceability. The findings reveal performance trade-offs and highlight how data structure selection influences the efficiency and clarity of algorithmic problem-solving. This project illustrates the practical application of dynamic arrays, recursion, and tree structures within a logic-based, constrained environment.**

## I. Literature Survey

In order to develop a Sudoku game from scratch, we need to examine various algorithmic approaches in terms of their space and time complexities and efficiency of the algorithm. For the Sudoku game in particular, we further need to explore the ultimate method which could be used for the implementation of the automated Sudoku solver. We selected three articles which delve deep into the use of recursive backtracking and brute-force methods in the context of Sudoku solving.

The first article, *"An implementation of backtracking algorithm for solving a sudoku-puzzle based on Android"* by Kusuma et al. (2020), discusses the implementation of a Sudoku solver using the backtracking algorithm that is compatible with the Android operating system. Their aim was to study how to optimize the backtracking algorithm despite possible hardware constraints. The researchers concluded that, although the backtracking approach is computationally exhaustive, it could be executed competently if the algorithm was well structured and integrated with constraint-checking functionality to prune the search space. We found this article beneficial as it discusses how to use the backtracking approach for interactive platforms like mobile games, in a way that is adaptable to all devices despite hardware limitations.

The second article, *"Analysis of the concept of solving the Sudoku game using backtracking and brute force algorithms"* by Oktaviani et al. (2023), analyzes backtracking and brute force in terms of efficiency. The researchers found that the efficiency of the brute force approach is directly proportional to the level of difficulty of the puzzle—becoming computationally impracticable as difficulty increases. This is because brute force tries every possible combination of numbers until the correct one is found, performing many redundant steps without early error detection. In contrast, the backtracking approach offers quicker detection of invalid paths by checking constraints (row, column, and subgrid uniqueness) after each number is placed. Although brute force is simpler to implement, backtracking is significantly more efficient. This led us to exclude brute force as a viable option and focus entirely on backtracking.

The third article, *"Recursive backtracking for solving 9x9 Sudoku puzzle"* by Senthilnathon and Gopal (2016), focuses specifically on how recursive backtracking works. It examines recursion depth, time taken to solve puzzles of varying difficulty, and its ability to detect invalid moves early. It also addresses edge cases such as puzzles with no solutions or multiple solutions, offering practical insights into structuring the recursion process safely and effectively. Their emphasis on recursion management helped us design our algorithm to prevent logical errors when an invalid path is encountered.

Collectively, these three articles helped us decide to use recursive backtracking as the foundation for our automated Sudoku solver. They also provided useful guidance for structuring the algorithm and planning our experiment to evaluate its efficiency and performance.

## II. Experiment Design

### *Objectives*

The primary goal of this experiment is to evaluate the efficiency and correctness of the implemented Sudoku solver using the recursive backtracking algorithm. The solver's performance is assessed across multiple difficulty levels and under different puzzle conditions, including unsolvable or ambiguous configurations. The experiment design draws inspiration from the methodologies proposed by Oktaviani et al. (2023), Senthilnathan & Gopal (2016), and Kusuma et al. (2020).

- To measure the performance of the recursive backtracking algorithm in terms of execution time across easy, medium, and hard Sudoku puzzles.
- To assess correctness, ensuring that the solver consistently produces valid solutions for solvable puzzles and appropriately handles unsolvable cases.
- To compare the results with expectations from the literature and identify patterns in performance and behavior based on puzzle complexity.

*Experimental Setup*

- **Platform**: The Sudoku game and solver will be run on a standard desktop or laptop environment using C++.
- **Input Data**: A total of 15 Sudoku puzzles will be tested:
  - 5 Easy puzzles
  - 5 Medium puzzles
  - 5 Hard puzzles

  Difficulty levels are selected from established datasets such as Project Euler, SudokuWiki, or Kaggle.
- **Solver Algorithm**: A recursive backtracking algorithm is used as the baseline solver. The algorithm performs constraint checking after each number is placed, ensuring that the value does not violate row, column, or 3x3 subgrid rules.

*Variables*

- **Independent Variable**: Puzzle difficulty level (Easy, Medium, Hard)
- **Dependent Variables**:
  - Execution time (measured in milliseconds)
  - Solver correctness (evaluated as correct/incorrect based on whether the final solution is valid)

*Procedure*

- **Initialization**: Each puzzle is loaded into the SudokuBoard class, and the timer is started.
- **Execution**: The solver applies recursive backtracking, attempting values from 1–9 in each empty cell while validating constraints at each step.
- **Data Collection**: After solving, the following metrics are recorded:
  - Total execution time
  - Whether the final solution is valid

## Results Analysis

*Execution Time Comparison*

The execution times for the Sudoku solver across all difficulty levels (Easy, Medium, and Hard) were consistent with the expected patterns outlined in the literature. As the difficulty level increased, the execution time also increased, reflecting the greater complexity and constraint checking required.

*Easy Puzzles:* The solver consistently solved the puzzles in the range of 0.01966 ms to 0.02873 ms, reflecting minimal backtracking and efficient constraint validation. This is within the expected range and demonstrates the solver's ability to handle easy puzzles quickly.

- **Puzzle 1**: Solved in 0.01966 ms — Extremely fast, likely minimal backtracking.
- **Puzzle 2**: Solved in 0.02319 ms — Still very fast, efficiently solved.
- **Puzzle 3**: Solved in 0.01744 ms — Very fast, minimal backtracking.
- **Puzzle 4**: Solved in 0.01990 ms — Fast, minimal backtracking or early constraint satisfaction.
- **Puzzle 5**: Solved in 0.02873 ms — Slightly more time, still very fast.

*Medium Puzzles:* The solver's times ranged from 0.02728 ms to 0.04121 ms, slightly longer than Easy puzzles but still efficient. The slight increase in time is expected as Medium puzzles often require more backtracking and constraint checking.

- **Puzzle 1**: Solved in 0.02728 ms — Fast, still efficient for a medium puzzle.
- **Puzzle 2**: Solved in 0.02939 ms — Slightly more time, but still very quick.
- **Puzzle 3**: Solved in 0.03750 ms — Slightly more time, possibly more branching or constraint checks.
- **Puzzle 4**: Solved in 0.02911 ms — Efficiently solved, similar to earlier puzzles.
- **Puzzle 5**: Solved in 0.04121 ms — A bit more time, still efficient for medium difficulty.

*Hard Puzzles:* The solver took 0.04198 ms to 0.05738 ms to solve the Hard puzzles. This is within the expected range for difficult puzzles, where backtracking and constraint propagation require more computational resources. The increase in time is consistent with the higher complexity and more rigorous checks needed to ensure a valid solution.

- **Puzzle 1**: Solved in 0.04198 ms — Efficiently solved, a bit more time as expected for Hard.
- **Puzzle 2**: Solved in 0.05738 ms — Slightly more time, likely due to additional backtracking or constraints.
- **Puzzle 3**: Solved in 0.05451 ms — Similar to Puzzle 2, efficient with slightly more time spent.
- **Puzzle 4**: Solved in 0.04590 ms — Efficient, with minor increase in time compared to earlier puzzles.
- **Puzzle 5**: Solved in 0.05023 ms — A bit more time, still within expected range for Hard.

*Correctness*

The solver correctly solved all puzzles, as expected. Each solution adhered to Sudoku rules, with no incorrect or invalid solutions. This shows that the backtracking algorithm successfully solved the puzzles across all difficulty levels.