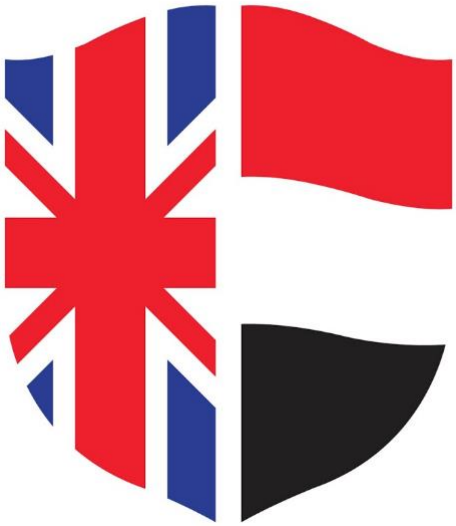


23CSCI05I

Logic and Artificial Intelligence

Phase 2 Report

Sudoku Puzzle Solver

Team Members:

ID	Name	Email
224030	Amir Mohamed	amir224030@bue.edu.eg
228074	Amr Magdy	amr228074@bue.edu.eg
218767	Sameh Ayman Makram	sameh218767@bue.edu.eg

Abstract:

The project aims to create an intelligent Sudoku puzzle solver utilizing two distinct algorithms: Backtracking Algorithm and the Genetic Algorithm. Sudoku is one of the most popular logic puzzles that given a 9x9 grid, digits 1-9 must be filled in the grid so that the result is a unique total of nine digits for each row, column, and 3x3 sub grid. This project aims to create a user-friendly application that implements both backtracking and genetic algorithms to both teach and solve the sudoku puzzle of varying difficulties, walking through step-by-step the player how it is resolved in the process.

Introduction:

Sudoku is a puzzle game invented in the USA in 1979 where it offered entertainment through a challenging and innovative blend between logic and strategy. Consequently, it has been an interesting topic of the implementation of various algorithms that not only solve the puzzle on a wide spectrum of difficulty but also serve as educational tools for thought-provoking puzzle analysis. As such, this project delves into the underlying mechanisms of a sudoku puzzle through the implementation of backtracking algorithm and genetic algorithms, two popular algorithms used in solving sudoku puzzles. Backtracking is a powerful algorithm that thoroughly examines all possible paths until a valid solution is found, presenting a clear and systematic approach to solving problems. If a solution were to deviate, the algorithm proceeds in reverse and investigates substitute routes. As a result, it works especially well on tasks under specific set of constraints like Sudoku [1]. This approach offers a clear step-by-step process, making it an ideal tool for learning the underlying strategies that take place when solving a Sudoku puzzle [2]. In contrast, genetic algorithms that derive their naming from the idea of natural selection serve as a capacity enhancer and imitate the random breeding strategies characteristic of the natural process. The algorithm operates on each of these solutions that evolve into chromosomes by excruciating process which includes selecting, crossing-over and mutating in order to generate the best-fit solution [3]. In spite of the overall effectiveness of such algorithms, their efficiency can be dulled by the

problem itself and the overall level of difficulty of the puzzle at hand [5]. Overall, the goal of this project is to put to a test both the strengths and the limitations of said algorithms with the end in view of demonstrating the full extent to which they can provide a solution and translate the strategy in a visual form with the audience not only finding a solution but also getting familiar with the logic that leads to it.

Problem Statement

Sudoku is a popular logic and puzzle game that can escalate rather quickly in difficulty and be a time-consuming and challenging task. Since there are proven ways of solving sudoku puzzles efficiently regardless of difficulty, this project aims to create an automated tool that can efficiently solve Sudoku puzzles given the sudoku puzzle at hand.

Scope:

- Allow users to choose among predefined sudoku puzzles and decide on the size of the overall puzzle.
- Allow users to choose between either backtracking or genetics algorithms to solve the puzzle.
- Show performance metrics such as speed, steps and iterations needed to reach the right solution.
- Display the game in professionally made UI to improve accessibility and mitigate any difficulties that might be faced in the learning process.

Limitations:

- The sudoku solver may struggle with complex puzzles, which will consume an exorbitant amount of time to solve the puzzle, hence affecting the overall performance.
- In the case where the user inputs a sudoku puzzle, he or she may input an invalid puzzle, increasing the complexity in terms of error handling for the solver.

Literature Review

1. Backtracking vs Genetic Algorithms for Sudoku Solving: A Comparative Analysis

In the research paper, A.A. Abd El-Rahman et al used both backtracking and naïve bayes to solve sudoku puzzles, highlighting both their strengths and weaknesses in terms of efficiency and accuracy. According to the paper, backtracking has a higher efficiency output in terms of time as it relies on exploring various solutions until it reaches the desired one where it stops executing. The study shows that the backtracking algorithm is a very effective method for solving Sudoku puzzles. It is a fast and efficient technique that can find solutions for puzzles of any difficulty level. Backtracking can also be useful in statistical tests to compare different methods [1].

The approach is as follows:

1. Find an empty square.
2. Try placing digits 1 to 9 in that square.
3. Check if the placed digit is valid according to the Sudoku rules.
    a. If the digit is valid, recursively try to fill the remaining board using steps 1-3.
    b. If the digit is invalid, go back to the previous step and try a different digit in the same square.

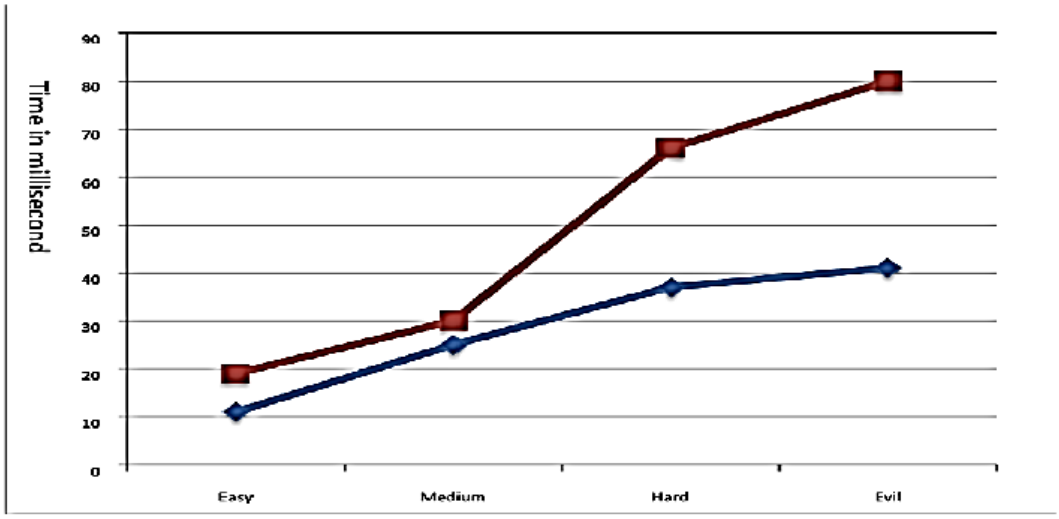


Fig 4.2 Running time of naïve bayes and backtracking algorithm

2. Human Problem Solving: Sudoku Case Study

Focuses on back tracking algorithm as model solve Sudoku puzzle, we can say also that brute-force approach to solving CSP is called backtracking. Backtracking technique to solve Sudoku puzzle work as the following:

Firstly, the backtracking search begins with an empty variable assignment as the state and then it attempts to find a solution iteratively by assigning values to variables one by one. When it discovers the constraint violation, it gives up the solution and restarts the search by backtracking. In such a way, it goes over the tree of possible partial assignments [2].

In detailed way I list the backtracking mechanisms that mentioned in the paper 2 in 7 main steps as the following:

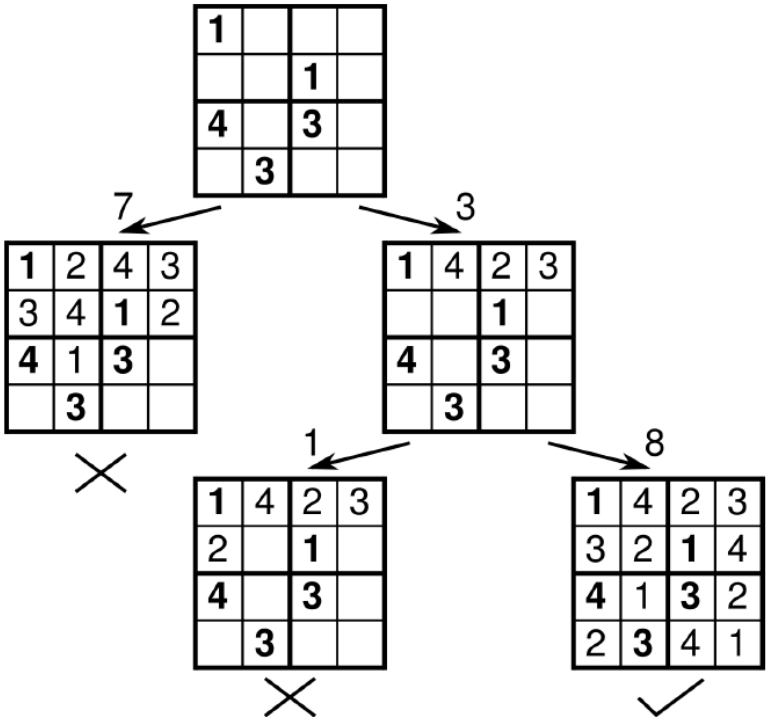
1. Start with an empty assignment of values to variables as an initial step.
2. Choose a variable to assign a value (valid value) to. This can be done in various ways, such as selecting the variable with the fewest possible values or the most constrained variable (the one with the fewest legal values based on the current

assignment) so assigning a value to a variable depends on more than one factors as we can notice.

3. Choose a value from the domain of the selected variable and assign it to the variable.
4. Check if the assignment violates any constraints, in other word check if the assigned value Not valid. If it does, go back to step 3 and choose a different value for the variable, going back to step 3 this what called backtracking.
5. If the assignment does not violate any constraints, move on to the next variable and repeat steps 2-4.
6. If all variables have been assigned values and the assignment satisfies all constraints, a solution has been found. If not, go back(backtrack) to the previous variable and choose a different value for it.
7. Repeat steps 2-6 until a solution is found or all possible assignments have been tried.

Backtracking works by systematically exploring the search space of possible assignments. It uses a depth-first search strategy, meaning it explores(travers) one branch of the search tree as far as possible before backtracking and trying a different branch. This allows it to efficiently search through the large number of possible assignments in a CSP.

In paper 2 case, backtracking starts with an empty grid and assigns values to the empty cells one by one. At each step, it checks if the assigned value violates any of the Sudoku rules. If a violation is found, it backtracks and tries a different value for the previous cell until find valid solution.



Consider the propagation of simple 4*4 Sudoku puzzle:

1	2,4	2,4	2,3, 4
2,3	2,4	**1**	2,3, 4
4	1,2	**3**	1,2
②	**3**	2,4	1,2, 4

3. A Novel Hybrid Genetic Algorithm for Solving Sudoku Puzzles

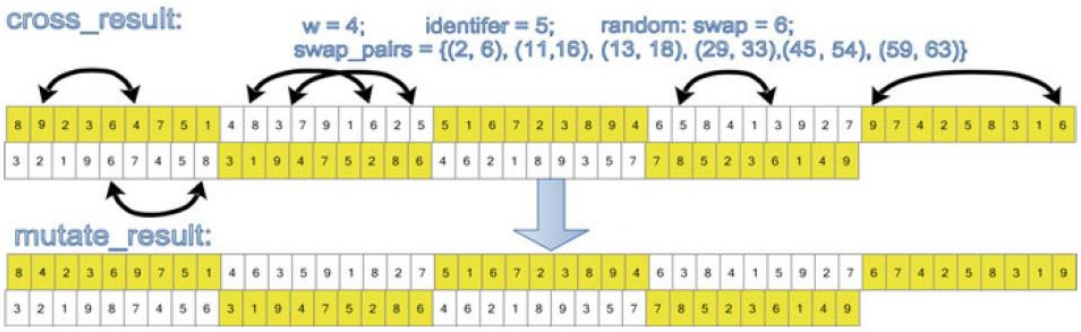
This paper by Xiu Qin Deng and Yong Da Li proposes a unique strategy for solving Sudoku puzzles using a hybrid genetic algorithm. The authors recommend modifications to the selection operator, crossover operator, and mutation operator of the genetic algorithm specifically for Sudoku puzzles.

The paper begins with an introduction to genetic algorithms (GAs) and particle swarm optimization (PSO) algorithms, which are widely used in computer science and engineering. The authors aim to determine if a genetic algorithm can effectively solve Sudoku puzzles. The paper provides an example Sudoku puzzle and its corresponding solution. The fitness function for the genetic algorithm is based on the rules that a valid Sudoku solution must follow. This function calculates the number of duplicate numbers in each row and column and assigns a fitness score based on these violations. A lower score indicates a better solution. Chromosomes (potential solutions) for the Sudoku puzzle are represented using a combination of 81 numbers. Numbers are obtained from right and left to base coding, with zeros representing empty cells and non-zero numbers representing given numbers in the problem. The encoding is divided into 9 sub-blocks, each corresponding to a 3x3 sub-grid. The selection operator of the genetic algorithm is improved by dividing the population into groups based on fitness scores [3].

$$w = \sum_{i=1}^{9} (r_i + c_i)$$

Chromosomes (potential solutions) for the Sudoku puzzle are represented using a combination of 81 numbers. Numbers are obtained from right and left to base coding, with zeros representing empty cells and non-zero numbers representing given numbers in the problem. The encoding is divided into 9 sub-blocks, each corresponding to a 3x3 sub-grid. The selection operator of the genetic algorithm is improved by dividing the population into groups based on fitness scores. Chromosomes with lower fitness scores have a higher probability of being selected as parent candidates. This approach aims to increase the diversity of selected chromosomes and improve the chance of selecting

chromosomes with more diverse genetic information. The crossover operator utilizes a random number of crossover points to exchange sub-blocks between two selected chromosomes. The information exchange pattern of bit-shift optimization is used to make the crossover operator more directional. The mutation operator swaps two elements within a sub-block to maintain the constraint that each sub-grid must contain each number once.



4. Paper: A study of Sudoku Solving Algorithms

This thesis explores three different algorithms for solving Sudoku puzzles - backtrack, rule-based, and Boltzmann machines. The focus is to measure and analyze the solving potential of these algorithms. However, additional aspects covered include difficulty rating of puzzles, Sudoku puzzle generation, and how well the algorithms are suited for parallelization.

The goal is to evaluate how each algorithm performs across these areas and draw comparisons between them. The thesis also aims to make general conclusions about Sudoku puzzles based on studying these algorithms [4].

The following is a brief description of each algorithm and its pseudo code:

- Backtrack: A brute-force approach that tries different values and backtracks when a choice doesn't work. It guarantees a solution if one exists but may take

```
function Backtrack(puzzle):
    (x, y) = findEmptySquare(puzzle)  // Find a square with the fewest candidates
    if (x, y) == null:  // All squares filled, puzzle solved!
        return puzzle

    for value in puzzle[y][x].possibilities():
        puzzle[y][x] = value  // Try assigning a value

        result = Backtrack(puzzle)  // Recursively solve the modified puzzle
        if result is not null:  // If the recursion found a solution
            return result

        puzzle[y][x] = 0  // Reset the value (backtracking)

    return null  // No solution found
```

exponential time.

- Rule-based: Applies logical rules derived from human-solving strategies to fill cells or eliminate possibilities. This heuristic might not solve all puzzles and is combined with a brute-force component (guessing). The set of rules are the following:

    - Naked Single: If a square has only one possible number remaining, that number must be placed there.
    - Hidden Single: If a number can only fit into one square within a region (row, column, or box), that number must be placed there.
    - Naked Tuple (Pair, Triple, etc.): If a group of squares within a region has the same limited set of candidates (e.g., only candidates 3 and 7), then those candidates can be eliminated from all other squares in that region.

- Hidden Tuple (Pair, Triple, etc.): If only a specific group of squares within a region can hold a set of candidates, those candidates can be removed from all other possible locations within those squares.

```
puzzle Rulebased(puzzle):
    while(true):
      //Apply the rules and restart the loop if the rule
      //was applicable. Meaning that the advanced rules
      //are only applied when the simple rules failes.
      //Note also that applyNakedSingle/Tuple takes a reference
      //to the puzzle and therefore changes the puzzle directly
      if(applyNakedSingle(puzzle)) continue
      if(applyNakedTuple(puzzle)) continue

      break
    }

    //Resort to backtrack as no rules worked
    (x,y) = findSquare(puzzle) //Find square with least candidates
    for i in puzzle[y][x].possibilities() //Loop through possible candidates
    puzzle[y][x] = i //Assign guess
    puzzle' = Rulebased(puzzle) //Recursion step
    if(isValidAndComplete(puzzle')) //Check if guess lead to solution
    return puzzle'

    //else continue with the guessing
    return null //No solution was found
```

- Boltzmann machine: Uses a constraint-solving artificial neural network. Sudoku constraints are encoded into network weights and solved to reveal a valid solution.

Deterministic algorithms (backtrack and rule-based) proved superior, exhibiting precise execution times, and generally outperforming the Boltzmann machine – particularly with harder puzzles. The Boltzmann machine's stochastic nature led to higher execution time

variance, and it struggled within the given timeframe. Temperature descent methods significantly impacted the Boltzmann machine's success.

5. Genetic Algorithms and Sudoku

The paper discusses using genetic algorithms (GAs) to solve Sudoku puzzles. Traditional backtracking algorithms work well for classic 9x9 Sudoku grids but struggle with larger grid sizes since the general problem is NP-complete. GAs, which are biologically inspired optimization algorithms, seem like a promising approach but turn out to be remarkably ineffective for Sudoku puzzles.

The author implemented a GA-based Sudoku solver in C++. It encodes solutions as permutations of digits within sub grids, uses a fitness function based on counting duplicate digits in rows/columns, and employs genetic operators like crossover and mutation tailored for the problem. The GA is restarted when stuck in local optima by creating new populations from accumulated best solutions.

Despite tuning parameters like population size and using techniques like predetermined square filling, the GA's performance on Sudoku puzzles was disappointing. It often got stuck in local optima and required many restarts to find the correct solution, if at all. Changing GA parameters had unpredictable effects across different puzzle instances.

The author conjectures that Sudoku may be an inherently GA-hard problem due to the discrete nature of the solution space and prevalence of local optima with the used fitness function. A finer-grained fitness function could potentially improve GA performance [5].

Similar Implementations:

Link: https://tirl.org/software/sudoku/

Automatic Sudoku Solver includes a Sudoku solver that uses Javascript and is easily handled by the user. However, it might not be the most efficient, averaging seconds for most puzzles and up to 5 minutes for difficult puzzles due to the sluggishness of browsers in terms of Javascript performance. Lastly, "Turbo" mode can be faster in some browsers, but its risk of failure might not be worth it.

Link: https://anysudokusolver.com/

AnySudokuSolver is an online free platform that provides solutions of Sudoku puzzles in mere seconds. It utilizes different algorithms, such as brute force and dancing links algorithm, with various speeds in finding the solution. Using it simply requires entering all the initial numbers by the user in accordance with Sudoku rules and hitting “Next” followed closely by "Solve" and in a blink of an eye, the website will send the correct answer for the puzzle.

Link: https://sudokuspoiler.com/sudoku/sudoku9

Sudokuspoiler is another online sudoku solver which provides the user with the option to among 50 different solvers. The user can input his or her own numbers and, upon completion, can click "Solve" to get the final solution for said puzzle. If the user would only want the solution of one cell, they can easily do so by clicking the "Solve Cell" button. The website can list up to 10 solutions and guide the user across them. In case of an error, the user may click "Unsolve” to get the wrong answers to be deleted and try again. Lastly, there is the "Reset" button to start from scratch.

Dataset:

Link: https://www.kaggle.com/datasets/rohanrao/sudoku

This dataset consists of 9 million Sudoku puzzles and their solutions. Each puzzle is presented in the form of a single row in which the 0-digit denotes the empty cell, and 1-9 digit defines the filled cell. This information can be exploited for the training of AI models that do some Sudoku puzzles or some of their analysis. However, although the number of recorded puzzles doesn't indicate the real number of potential puzzles, this collection can be used for different purposes.

Algorithms Used: Critical Analysis and Justification

- **Backtracking:**

  Offers a systematic and efficient method for finding a solution, particularly for well-defined puzzles. If a proposed solution violates established constraints, the algorithm backtracks and explores alternative paths. This makes it particularly well-suited for constraint satisfaction problems like Sudoku, where adherence to specific rules is paramount. However, it can become computationally expensive for highly complex puzzles as it faces challenges in scenarios with a high degree of branching or an expansive search space, resulting in increased computational time.

- **Genetic Algorithms (GA):**

  Provides a more flexible approach, exploring a wider range of potential solutions and potentially finding unique solutions Backtracking might miss. They do so by mimicking the process of natural selection, where solutions evolve over successive generations through selection, crossover, and mutation operators. GAs are particularly effective in optimization problems, puzzle-solving scenarios, and in situations where the search space is not well-defined. However, it may require more iterations to converge on an answer compared to Backtracking and is often times accompanied with increased inefficiency and slower output compared to backtracking methodologies. They may struggle to converge to optimal solutions,

and the quality of solutions heavily depends on the design of genetic operators and the representation of the problem domain.

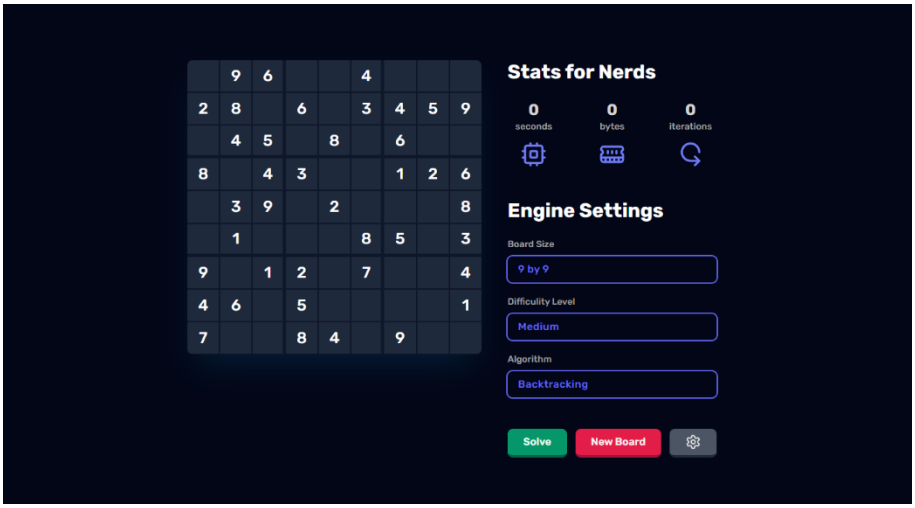
- **Comparison:**

The backtracking method is one of the most efficient pragmatic ways which provides a high percentage of finding a solution for traditional Sudoku puzzles having only one solution. It accomplishes this by the process of scanning through possible solutions and hence, looking for exit via following Sudoku rules which makes it a trustworthy strategy. However, despite working efficiently with simple puzzles that have few options for the way out, it will often fail when trying to resolve complicated puzzles that have numerous branches for the solutions.

Unlike genetic algorithms, which are highly adaptable, the latter does not provide such adaptability. genetic algorithms study areas much wider and may combine to have a solution to the problem that backtracking is not capable of. It is this case that puzzles with several answers or alternative strategies are usually the most appropriate for this method. On the other hand, genetic algorithms can be computationally consuming because of the higher cycles of processing, and it may land into local optima as well.

Lastly, the two methods discussed, i.e. Backtracking and genetic algorithms, are the most suitable ones for solving normal sudoku puzzles. Retracing is more reliable than using genetic algorithms. Genetic algorithms are better in giving different variants. This productivity dispute between the two algorithms, the consequence of their unique structure, is what makes them have a different design. The untapped backtracking is a methodical approach, and hence, the outcome is not to employ effort unnecessarily. The algorithm of genetic needs to evaluate the soma wide solution pool, it therefore requires greater computational power and time to achieve similar results to the backtracking algorithm.

Project Development:



Our goal was to develop a sudoku solver that relies on the different AI search algorithms discussed earlier in the report. We opted for a friendly user interface that showed metrics such as: run time, memory consumed and number of iterations that varied for each board, difficulty, and algorithm.

The project was implemented using the following tech stack:

- **Python:** Python was the language of choice that allowed for simple implementation of both the backtracking and genetics algorithms, as well as the logic for the board puzzle generation.
- **Electron:** is an open-source framework that allows for the development of desktop applications using web technologies such as HTML, CSS, and JavaScript. It was the method of choice for the GUI implementation due to its faster development and ability to run on different operating systems such as Windows, MacOS and Linux.

Features:

The desktop application allows the user to choose boards of different sizes (4x4, 6x6 and 9x9). Additionally, users can choose between four different difficulty levels: easy, medium, hard, and very hard. These levels remove 30%, 50%, 70% and 90% off the board, respectively. After a board size is chosen and difficulty set, the user can decide which of the two algorithms to choose from. If the user chooses backtracking, he or she could proceed by clicking solve and seeing the board solved and the metrics updated accordingly. If the user chooses the genetic algorithm, a slider will appear, allowing them to specify the starting population size for the algorithm.

Documentation:

BacktrackingSolver:

This class implements the backtracking algorithm.

Attributes:

- self.iterations (int): total number of times it tried placing a number in a cell.
- elapsed_time (float): measures the time taken to solve the puzzle
- memory_used (float): tracks the amount of memory used by the solver.
- Snapshots (list): keeps copies of the board at different stages of solving.

Functions:

- is_valid(board, row, col, num, size)

  this function checked if placing the number num in the cell at row row and column col of the provided sudoku board board was a valid move. makes sure no collision occurs in a row, box or column.

- backtracking_algorithm(board, size)

  this function was the core solver. it used backtracking to explore all possible placements of numbers and find a solution.

GeneticSolver

This class implements the genetic algorithm. It works by maintaining a population of possible solutions and evolving them over generations to find the best solution that solves the board.

Attributes:

- board (str): flattened string representation of the unsolved sudoku board.
- target (str): flattened string representation of the solved target sudoku board.
- chromosome (list): current candidate solution as a list of numbers representing the board.
- fitness (int): fitness score of the current chromosome
- iterations (int): number of generations explored during the solving process (initialized to 0).
- elapsed_time (float): time taken to solve the puzzle
- memory_used (float): memory used by the solver
- snapshots (list): list of json strings containing snapshots of the solving process at each generation

functions:

- mutated_board: generates a random valid sudoku board (used for creating initial solutions and mutations).
- create_gnome: creates a chromosome (candidate solution) by filling the board with random valid numbers.
- mate: combines the chromosomes of two parent solutions to create a new offspring solution. it randomly selects genes (board parts) from either parent or creates new random mutations with a certain probability
- fitness: calculates the fitness score of a solution by counting the number of squares that differ from the target board
- genetic_algorithm: this function runs the genetic algorithm.
- add_snapshot: saves a snapshot of the generation number, current best solution and its fitness score.

- get_iterations: Returns the total number of generations explored during the solving process.
- get_elapsed: Returns the time taken to solve the puzzle (in seconds).
- get_memory: Returns the memory used by the solver (in megabytes).
- perpare_board: Takes an unsolved and solved Sudoku board as input and converts them into flattened strings for easier manipulation within the solver.

BoardGenerator

Class used to generate board and its solution.

Attributes:

- size (int): size of the sudoku board (default 9x9)
- complexity (str): difficulty level of the generated board (default 'easy').
- board (list): the generated sudoku board as a 2d list of integers (0 for empty cells).

Functions:

- generate(self) -> List[List[int]]: Generates a random unsolved Sudoku board based on the specified size and complexity.
- __find_empty_cell(self) -> Tuple[int, int]: Finds the next empty cell (row, col) in the board. Returns None, None if no empty cells are found.
- __is_valid(self, num: int, pos: Tuple[int, int]) -> bool: Checks if placing a number (num) in a specific position (pos) is valid according to Sudoku rules (considering row, column, and sub-grid).
- __solve(self) -> bool: Solves the Sudoku board using backtracking algorithm.

Test Solver:

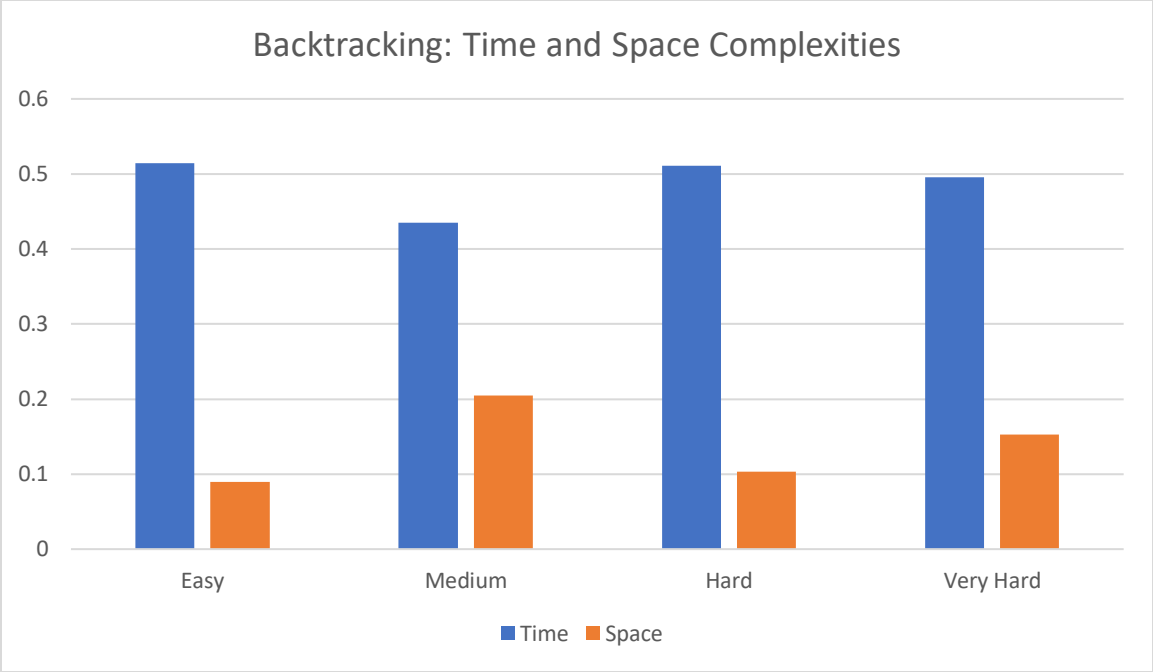
To evaluate our Sudoku solver's effectiveness, we employed a targeted testing approach. We compiled a limited dataset of Sudoku puzzles, ensuring a variety of difficulty levels. Each puzzle within this dataset came paired with its corresponding solved state. By feeding these puzzles into our solver, we were able to assess its accuracy. We compared the solutions generated by the solver against the pre-defined answers for each puzzle. This comparison allowed us to pinpoint any errors the solver might make in handling different Sudoku configurations and difficulty levels. This testing approach, while utilizing a limited set of puzzles, provided valuable insights into the solver's performance and its ability to correctly solve Sudoku puzzles.

Data and Metrics:

To quantitively assess the differences in performance of both algorithms, we analyzed the following metrics for 9x9 boards:

Backtracking

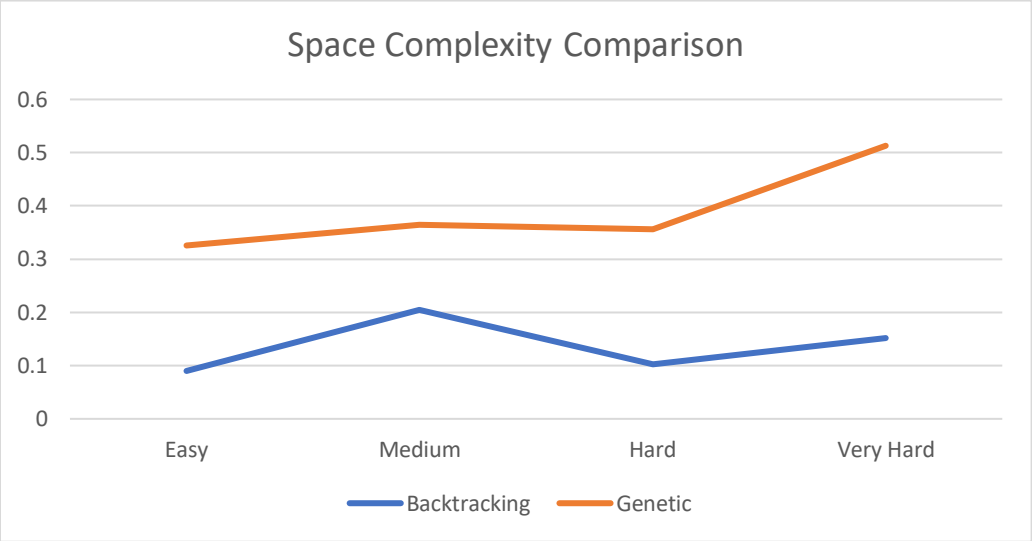
	Easy			Medium			Hard			Very Hard		
	Time	Space	Iterations	Time	Space	Iterations	Time	Space	Iterations	Time	Space	Iterations
Exp 1	0.482	0.18	106	0.447	0.211	108	0.592	0.059	143	0.449	0.141	101
Exp 2	0.56	0.059	124	0.395	0.2	93	0.413	0.125	93	0.536	0.195	123
Exp3	0.502	0.031	116	0.464	0.203	113	0.529	0.125	126	0.501	0.121	122
Avg	0.51467	0.09	115.33	0.4353	0.204667	104.67	0.51133	0.103	120.66	0.495	0.1523	115.33

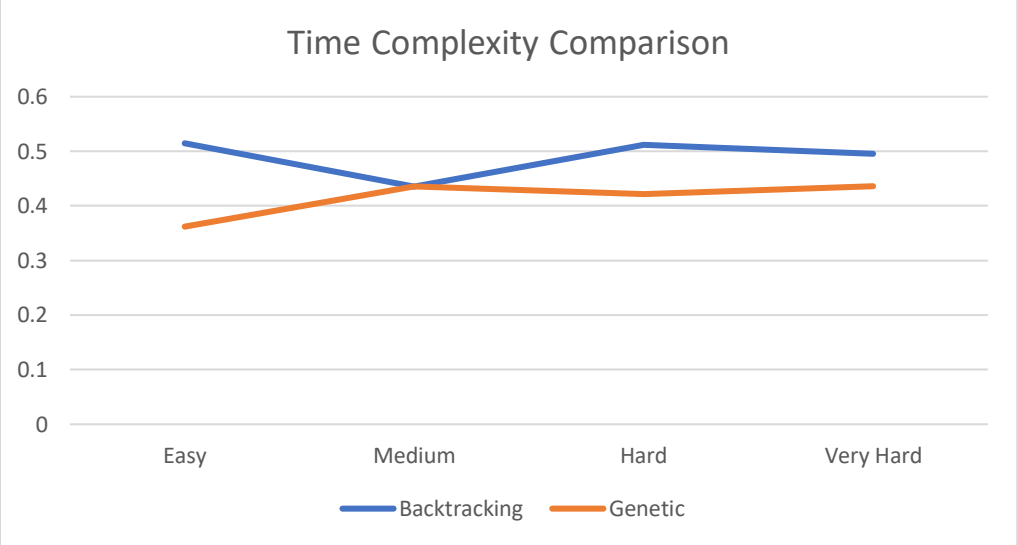


Data gathered revealed a significant advantage for the backtracking algorithm in terms of time complexity. As illustrated in the bar chart above, the average solving time for the backtracking algorithm remained relatively consistent across all difficulty levels for each board size.

Genetic Algorithm (250 Population Size)

	Easy			Medium			Hard			Very Hard		
	Time	Space	Iterations	Time	Space	Iterations	Time	Space	Iterations	Time	Space	Iterations
Exp 1	0.348	0.391	32	0.458	0.449	42	0.529	0.339	47	0.473	0.637	42
Exp 2	0.399	0.262	37	0.407	0.328	35	0.341	0.586	31	0.442	0.449	39
Exp3	0.339	0.324	30	0.441	0.316	36	0.394	0.145	42	0.393	0.453	36
Avg	0.362	0.325667	33	0.435333	0.364333	37.66667	0.421333	0.356667	40	0.436	0.513	39





Data shows that although the genetic achieves its goal in a better time on average with fewer iterations compared to backtracking algorithm, this advantage comes at a significant cost, which is memory consumption. The genetic algorithm requires storing and manipulating a population of candidate solutions throughout the evolutionary process and so depending on the population size, it may require greater memory to ensure sufficient exploration of the search space.

References:

[1] T. Navya, T. J. Mounika, S. Tharun, K. R, H. S, and B. A, “Comparative study on Sudoku using Backtracking algorithm,” *First International Conference on Smart Systems and Green Energy Technologies*, 2023, doi: https://doi.org/10.13052/rp-9788770229647.032.

[2] R. Pelánek, “Human Problem Solving: Sudoku Case Study,” 2011. Accessed: Mar. 27, 2024. [Online]. Available: https://www.fi.muni.cz/reports/files/2011/FIMU-RS-2011-01.pdf

[3] X. Q. Deng and Y. D. Li, “A novel hybrid genetic algorithm for solving Sudoku puzzles,” *Optimization Letters*, vol. 7, no. 2, pp. 241–257, Oct. 2011, doi: https://doi.org/10.1007/s11590-011-0413-0.

[4] P. BERGGREN and D. NILSSON, “A study of Sudoku solving algorithms” thesis, 2012. https://www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2012/rapport/berggren_patrik_OCH_nilsson_david_K12011.pdf

[5] J. M. Weiss, “Genetic Algorithms and Sudoku” thesis, 2009. https://micsymposium.org/mics_2009_proceedings/mics2009_submission_66.pdf