Sudoku Puzzle Solver

Project Introduction

Sudoku Puzzle Solver Using Backtracking and Genetic Algorithms

This project implements an **Sudoku Puzzle Solver** utilizing two distinct approaches:

1. **Backtracking Algorithm: A** systematic and recursive method to explore all possible puzzle configurations.
2. **Genetic Algorithm: An** evolutionary approach inspired by natural selection to find optimized solutions.

Sudoku is a well-known number-placement puzzle where a **9x9** (We also can try the 4x4 and 16x16) **grid must** be filled with digits (1-9), such that:

- Each row contains all digits without repetition.
- Each column contains all digits without repetition.
- Each **3x3 subgrid** contains all digits without repetition.

The solver efficiently tackles Sudoku puzzles by exploring possible configurations and finding the correct combination that satisfies all constraints.

Development Platform

Python 3.12.6
Tkinter for gui
Matplotlib.pyplot for visualization

Project Overview

The project is organized into multiple Python modules for modularity and clarity:

1. `backtracking.py`:
    a. Implements the Backtracking algorithm for Sudoku solving.
    b. Recursively fills the grid, backtracks on incorrect placements, and continues until a valid solution is found.
2. `Generic.py`:
    a. Genetic algorithms iteratively improve solutions by simulating natural selection, favoring fit individuals and introducing genetic variation through crossover and mutation.
3. `gui.py`:
    a. Provides a graphical user interface for the Sudoku Solver.
    b. Allows users to input puzzles, visualize solutions, and interact with the solver.
4. `main.py`:
    a. Serves as the entry point of the project.
5. `puzzle.py`:
    a. Contains functions to generate and take hints on Sudoku puzzles.
6. `utils.py`:
    a. The code checks if a given number can be placed at a specific location in a Sudoku grid by verifying that it doesn't conflict with any existing numbers in the **same row**, **column**, **or 3x3 "or any size given" subgrid**. The subgrid's starting index is calculated by subtracting the modulo of the row/column index by the subgrid size.

Why Backtracking and Genetic Algorithms?

Backtracking Algo

1. Initialization (Creating the Initial State)

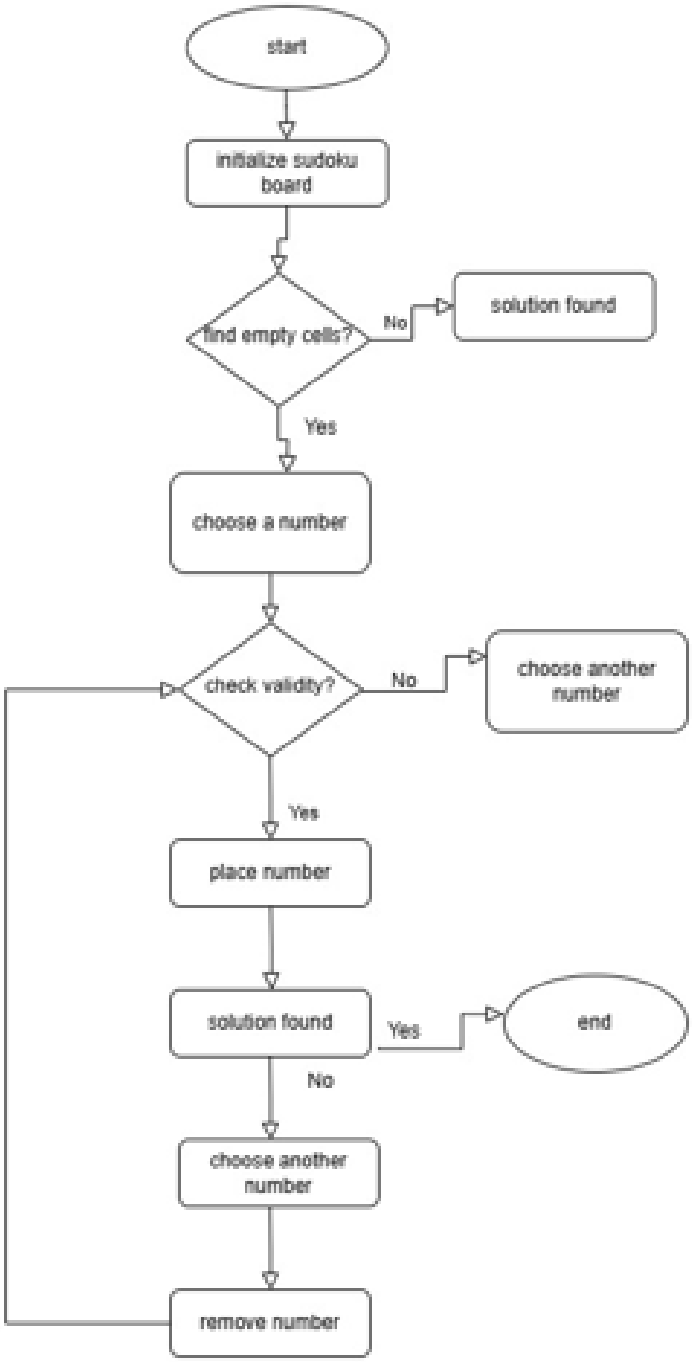
What happens?: The algorithm begins by setting up the Sudoku puzzle. The given puzzle grid is passed into the function, along with the list of empty cells that need to be filled. Several lists are initialized to help manage the solving process:

SL (Solution List): Keeps track of the cells that have been successfully filled.

NSL (Next Step List): Stores information about the next cell to attempt filling and tracks the numbers previously tried in that cell for backtracking purposes.

DE (Dead End): Tracks the cells where the algorithm has backtracked, keeping record of cells that the algorithm had to return to.

Why is it important? Initializing these variables allows the algorithm to track progress, manage backtracking, and decide the next cells to fill. It sets the stage for solving the puzzle step-by-step.


start
initialize sudoku board
find empty cells?
No
solution found
Yes
choose a number
check validity?
No
choose another number
Yes
place number
solution found
Yes
end
No
choose another number
remove number

2. Selection (Choosing the Next Cell)

What happens?: The algorithm looks at the list of empty cells (empty_cells) and selects the next one to work on. It picks a cell and starts testing different numbers that could fit in that cell, following the sequence of attempted numbers (starting from the last one it tried).

Why is it important?: By selecting the next cell to focus on, the algorithm proceeds in an organized manner. It ensures that no empty cells are skipped, and each one is filled in a structured order.

3. Checking Validity (Evaluating Potential Moves)

What happens? For each candidate number, the algorithm checks whether it can be placed in the current cell without violating Sudoku's rules (i.e., no duplicates in the same row, column, or 3x3 grid). The function check_location_is_safe performs this check.

Why is it important? This step ensures that only valid numbers are placed in the grid. If the algorithm tries to place an invalid number, it will not proceed and will need to backtrack to find another possibility.

4. Backtracking (Undoing Invalid Moves)

What happens? If the algorithm encounters a situation where it cannot place a valid number in the current cell, it backtracks. Backtracking means the algorithm removes the most recent cell from the solution list (SL), sets the cell back to 0, and tries a different number for the previous cells. This is done by popping the last entry from SL and pushing it into DE.

Why is it important? Backtracking is the core of solving the Sudoku puzzle. If a number placement doesn't work out, the algorithm can go back and attempt a different choice for earlier cells. This prevents the solver from getting stuck and allows it to explore alternative paths to a solution.

5. Updating the Grid (Placing a Valid Number)

What happens? Once a valid number is placed in the selected cell, the Sudoku grid is updated. The algorithm proceeds to the next empty cell and continues the process.

Why is it important? This step keeps the puzzle's state accurate and up to date. It also ensures the progress is reflected visually (if a GUI is used) and helps track the number of cells that have been filled correctly.

6. Termination (Solution or Failure)

What happens? The algorithm finishes when either:

All the empty cells are filled with valid numbers, and the puzzle is solved.

The algorithm has exhausted all possibilities and backtracked through all options, indicating that the puzzle is unsolvable.

Why is it important? The termination condition ensures that the solver stops when a solution is found or when it is clear that no solution exists. This avoids infinite loops and ensures that the algorithm completes its process efficiently

Genetic Algo

The Genetic Algorithm (GA) is an optimization technique inspired by natural selection. In the context of Sudoku, the algorithm evolves a population of potential solutions into a valid Sudoku grid. Below is the step-by-step process:

1. Initialization (Creating the Initial Population)

What happens?
A population of potential Sudoku solutions is generated. Each solution is a 9x9 grid where rows are filled randomly, respecting the fixed numbers in the puzzle.

Why is it important?
This step provides diverse starting points for the algorithm to explore the solution space.

2. Fitness Evaluation

What happens?
Each solution is evaluated based on its "fitness," which is determined by counting conflicts in the rows, columns, and 3x3 subgrids. A solution with fewer conflicts has higher fitness.

Why is it important?
This ensures that solutions closer to a valid Sudoku grid are favored, guiding the search toward the optimal solution.

3. Selection (Choosing Parents)

What happens?
The algorithm selects pairs of solutions (parents) from the population for reproduction, often using tournament selection, which favors higher-fitness solutions while maintaining diversity.

Why is it important?
Selection focuses the algorithm's efforts on promising areas of the solution space while retaining genetic diversity.

4. Crossover (Recombination)

What happens?
Two parent solutions are combined to produce a child solution. Rows from both parents are mixed, creating a new grid that inherits traits from both parents. Fixed numbers remain unchanged.

Why is it important?
Crossover introduces new combinations of genes, allowing the algorithm to explore solutions that neither parent could achieve alone.

5. Mutation (Random Variation)

What happens?
With a small probability, random changes are introduced to the child solution, such as swapping two numbers within a row while respecting fixed cells.

Why is it important?
Mutation prevents the algorithm from getting stuck in local optima by introducing new variations into the population.

6. Replacement (Forming the Next Generation)

What happens?
The best solutions from the current generation are carried forward (elitism), while new solutions (children) replace fewer fit solutions in the population.

Why is it important?
This ensures the population improves over generations while maintaining diversity.

7. Termination

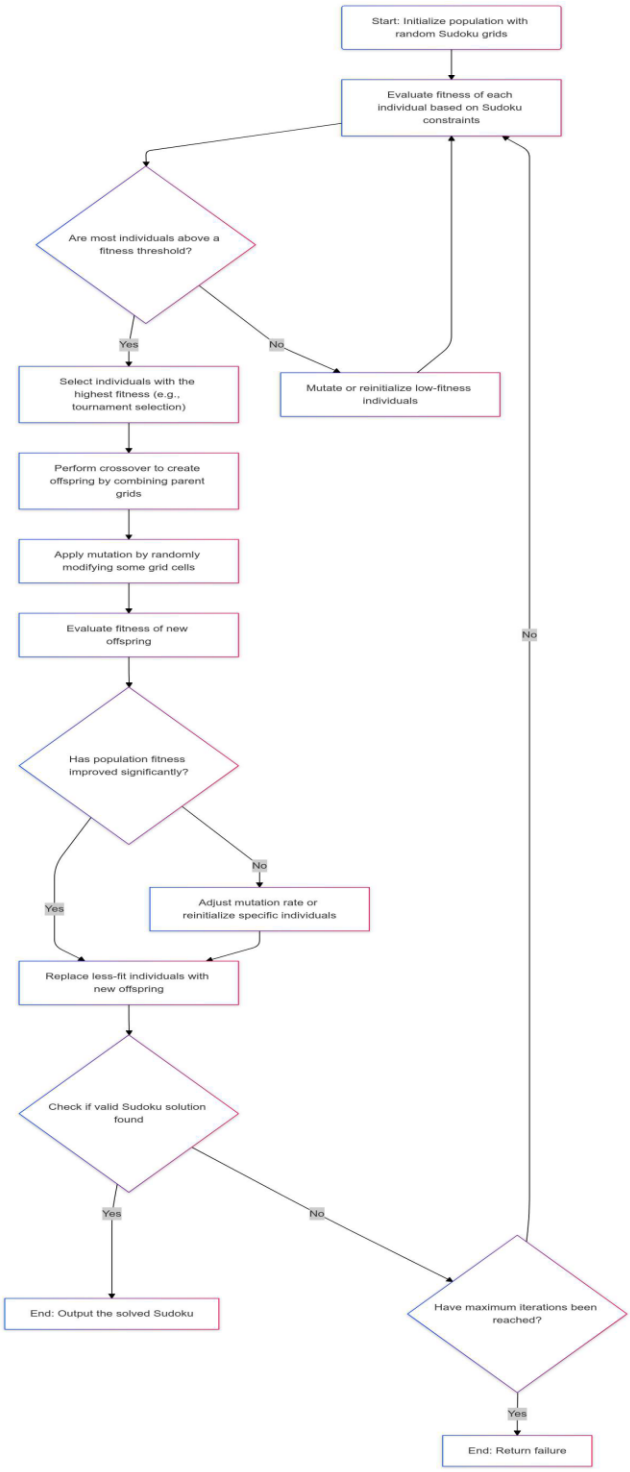
What happens?
The algorithm stops when one of the following conditions is met:

A valid Sudoku solution (fitness = 0 conflicts) is found.

The maximum number of generations is reached.

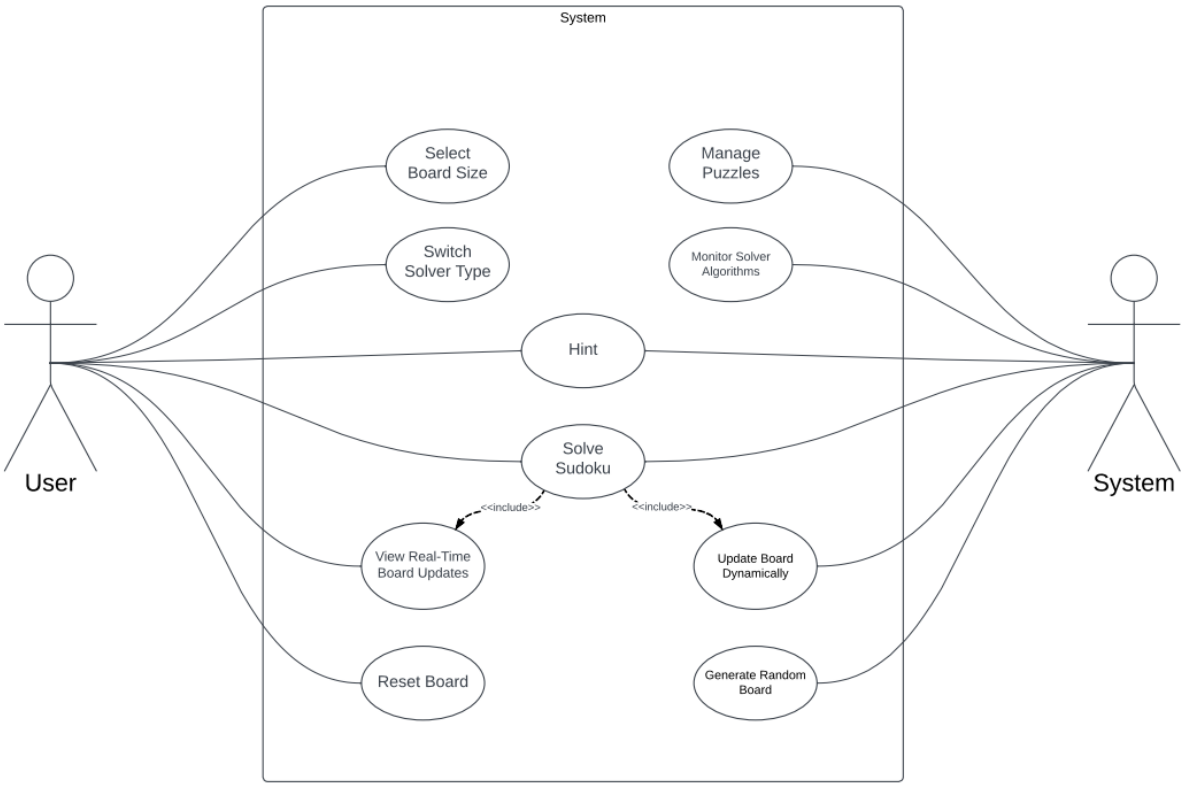
- Applications *(desktop, web, or mobile)* like the one you're developing, and what are the functionalities/features, and how they work *(if that information is available)*.

- A Literature Review of Academic publications *(papers/books/articles)* relevant to the problem you're trying to solve and the approach you're trying to implement *(at least 5 resources)*. You may find them by searching using Google Scholar.


Start: Initialize population with random Sudoku grids
Evaluate fitness of each individual based on Sudoku constraints
Are most individuals above a fitness threshold?
Yes
No
Mutate or reinitialize low-fitness individuals
Select individuals with the highest fitness (e.g., tournament selection)
Perform crossover to create offspring by combining parent grids
Apply mutation by randomly modifying some grid cells
Evaluate fitness of new offspring
Has population fitness improved significantly?
Yes
No
Adjust mutation rate or reinitialize specific individuals
Replace less-fit individuals with new offspring
Check if valid Sudoku solution found
Yes
No
End: Output the solved Sudoku
Have maximum iterations been reached?
Yes
No
End: Return failure

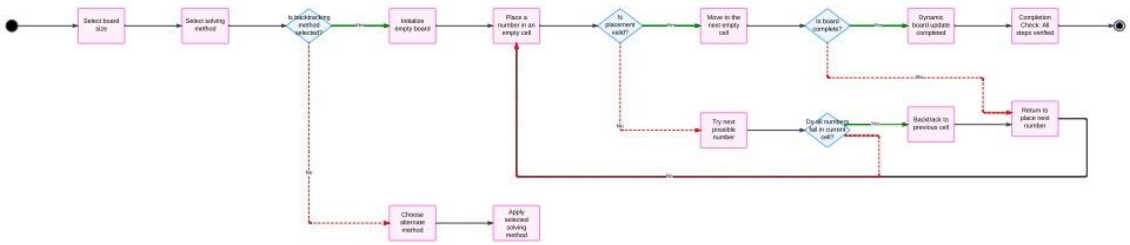
Proposed Solution & Dataset:

Use-Case Diagram:

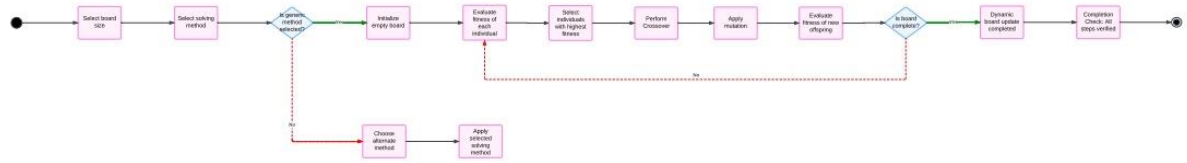


Activity Diagram:

backtracking

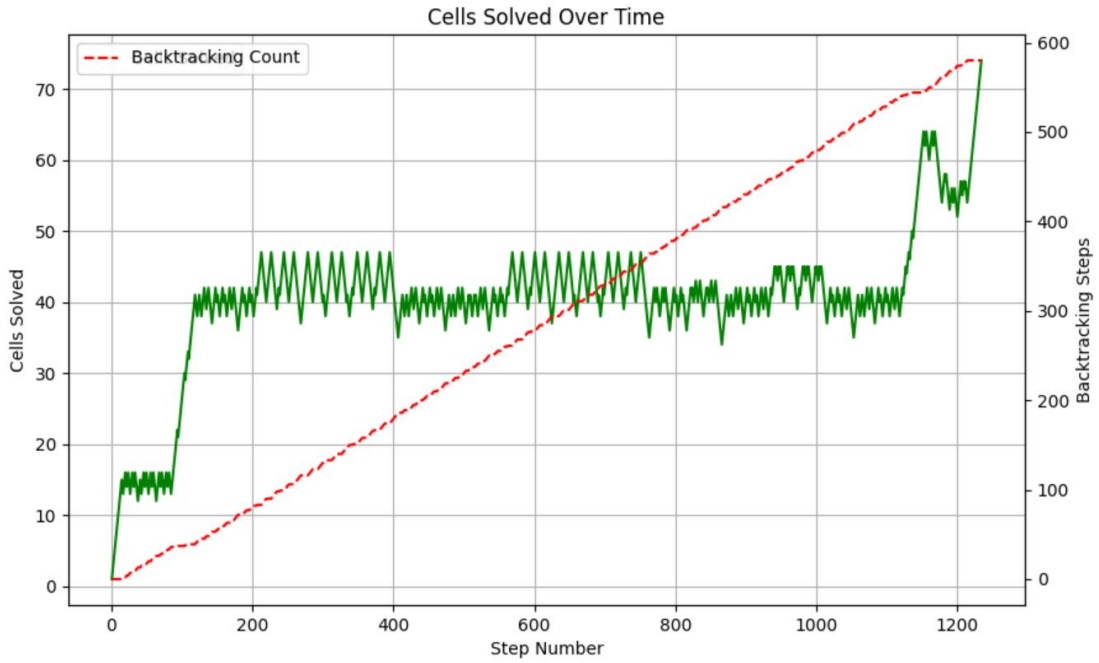


Genetic



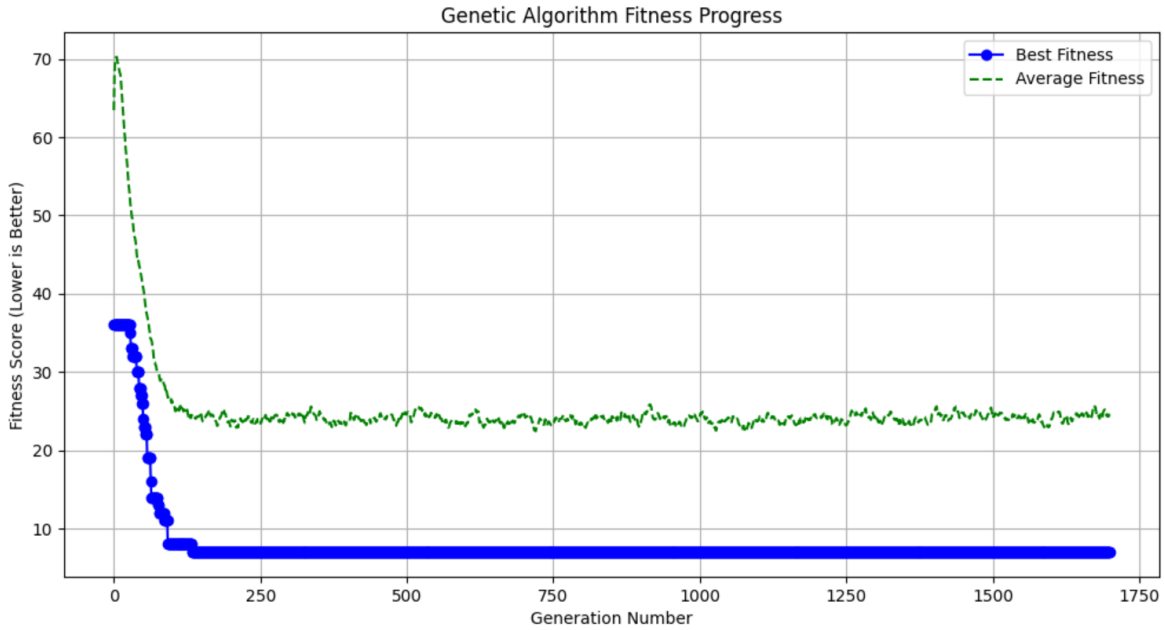
Analysis, Discussion, and Future Work:

Backtracking:



1. Initial Progress: Rapid progress with minimal backtracking (efficient start).
2. Mid-Progress: Stagnation due to conflicts, leading to heavy backtracking.
3. Late Progress: Sharp breakthrough, solving most cells quickly.
4. Final Stage: Solution stabilizes, backtracking plateaus.

Genetic



1. Early Progress: Rapid improvement in fitness score during initial generations.
2. Stagnation: Best fitness stabilizes around 7-8, while average fitness fluctuates (~25).
3. Observation: The algorithm finds near-optimal solutions early but struggles to improve further.
4. Summary: Convergence is efficient but limited; better mutation or selection strategies may help overcome stagnation.

Generation 0: Best Fitness = 36, Avg Fitness = 63.45

Generation 100: Best Fitness = 8, Avg Fitness = 26.20

Generation 200: Best Fitness = 7, Avg Fitness = 23.33

Generation 300: Best Fitness = 7, Avg Fitness = 24.46

Generation 400: Best Fitness = 7, Avg Fitness = 23.72

Generation 500: Best Fitness = 7, Avg Fitness = 24.63

Generation 600: Best Fitness = 7, Avg Fitness = 23.06

Generation 700: Best Fitness = 7, Avg Fitness = 24.61 Generation 800: Best Fitness = 7, Avg Fitness = 23.41

Generation 900: Best Fitness = 7, Avg Fitness = 23.80 Generation 1000: Best Fitness = 7, Avg Fitness = 24.32

Generation 1100: Best Fitness = 7, Avg Fitness = 23.85 Generation 1200: Best Fitness = 7, Avg Fitness = 24.11

- What are the advantages / disadvantages?

1. Genetic Algorithm (GA)

Advantages:

Exploration of Large Solution Spaces:

GA is excellent for searching large, complex, or poorly understood solution spaces.

It avoids local optima by using crossover and mutation, ensuring diversity in solutions.

Parallel Search:

Evaluates multiple solutions (population) simultaneously, which improves the chances of finding a global optimum.

Robustness:

Works well even when the problem is nonlinear, noisy, or has a high-dimensional solution space.

Scalability:

Can handle larger and more complex problems with appropriate parameter tuning.

Quick Initial Improvement:

As shown in the graph, GA rapidly improves in the initial generations, making it ideal for finding near-optimal solutions quickly.

Disadvantages:

No Guarantee of Global Optimality:

GA does not guarantee finding the global optimum, especially if the algorithm converges prematurely.

Parameter Sensitivity:

Performance depends on careful tuning of parameters (e.g., population size, mutation rate, crossover probability).

Computational Overhead:

Evaluating fitness for a large population can be computationally expensive, especially for complex problems.

Stagnation:

As seen in the Average Fitness curve, GA may plateau after early improvements, requiring techniques like restarting or adaptive mutations to escape stagnation.

Randomness:

Stochastic elements like crossover and mutation can make results inconsistent across runs.

2. Backtracking Algorithm

Advantages

Deterministic Results:

Provides consistent results for the same problem as it systematically explores all possible solutions.

Efficiency in Finite Spaces:

Works well for problems with small or finite solution spaces and clear constraints (e.g., Sudoku, N-Queens).

Simplicity:

Straightforward implementation compared to heuristic-based methods like GA.

Guaranteed Solution:

If a solution exists, backtracking will find it.

Memory Efficient:

Does not require maintaining a population or large data structures, as seen in GA.

Disadvantages

High Computational Cost:

As seen in the graph, backtracking steps grow steadily, making it inefficient for larger or more complex problems.

Requires exponential time in the worst case (e.g., if there are many constraints or no solution).

Inflexibility:

Struggles with unstructured or large solution spaces because it relies on deterministic traversal, which is prone to dead ends.

Plateaus in Progress:

As shown in the cells-solved graph, the algorithm may experience periods where no new progress is made due to constraint limitations.

Lack of Adaptability:

Not robust to noisy or dynamic problems; it is suited only for well-defined problem spaces.

Scaling Issues:

Poor scalability for problems with increasing dimensions or constraints, as backtracking grows exponentially.

Why the Algorithms Behaved This Way

Genetic Algorithm: Performed well initially due to exploration but stagnated as diversity decreased.

Backtracking: Progressed incrementally but slowed due to constraints and frequent backtracking.

Future Modifications

Genetic Algorithm:

Increase diversity (dynamic mutation, hybrid methods).

Use parallel populations or adaptive parameters.

Backtracking:

Improve constraint propagation.

Use smarter variable ordering or parallelism to reduce backtracking steps

Similar Applications

https://sudokuspoiler.com/sudoku/sudoku9

Sudoku Solver With Customize Grid Board Size.

https://www.sudoku-solutions.com/

This solver offers a number of features to help you improve your solving skills and practice solving strategies.

https://sudoku.com/sudoku-solver

Sudoku Solver Using Cutting-Edge Algorithms

Papers

6- https://www.riverpublishers.com/pdf/ebook/chapter/RP_9788770229647C32.pdf

This Paper Discusses The Development Of A Sudoku Solver Using MATLAB And Compares Different Algorithms, Highlighting The Efficiency Of Backtracking In Solving Sudoku Puzzles.

Comparative Study on Sudoku Using Backtracking Algorithm.

6- https://www.researchgate.net/publication/332371046_A_hybrid_backtracking_and_pencil_and_paper_sudoku_solver

This Research Introduces A Hybrid Solver That Combines Traditional Pencil-And-Paper Methods With Backtracking Algorithms, Achieving Efficient Solutions For Sudoku Puzzles.

A Hybrid Backtracking and Pencil and Paper Sudoku Solver.

6- https://micsymposium.org/mics_2009_proceedings/mics2009_submission_66.pdf
This Study Explores The Application Of Genetic Algorithms To Solve Sudoku Puzzles, Discussing The Performance And Challenges Compared To Standard Backtracking Algorithms.

Genetic Algorithms and Sudoku.

6- https://dl.acm.org/doi/10.1145/3194452.3194463
This Paper Presents A Genetic Algorithm With Modified Crossover And Mutation Operators Designed To Handle Local Optima, Enhancing The Efficiency Of Solving Sudoku Puzzles.

Genetic Algorithms with Local Optima Handling to Solve Sudoku.

6- https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group6Alexander/final/Patrik_Berggren_David_Nilsson.report.pdf
This Bachelor Thesis Examines Three Different Sudoku Solving Algorithms, Focusing On Their Solving Abilities And Efficiency.

A Study of Sudoku Solving Algorithms.

6- http://fendrich.se/blog/2010/05/05/solving-sudoku-with-genetic-algorithms/
This Article Shows a Python Library Made By **David Fendrich** For Genetic Algorithm Specifically For Solving Sudoku Puzzle.

Solving Sudoku With Genetic Algorithm by David Fendrich.