Università
Ca'Foscari
Venezia

Foundations of Artificial Intelligence
[CM0623-2]

Assignment Report

Constraint Propagation and Evolutionary Approaches Applied to Sudoku Solving Algorithms

Supervisor
Prof. Andrea Torsello

Student
Gianmaria Pizzo
Matriculation Number 872966

Academic Year
2024 /2025

Abstract

This project investigates two distinct AI-based algorithms for solving Sudoku puzzles, examining their performance on a large-scale dataset from Kaggle containing four million puzzles with varying levels of difficulty. The first algorithm combines constraint propagation and backtracking, enhanced with forward checking, the Minimum Remaining Values (MRV) and Least Constraining Value (LCV) heuristics, and Maintaining Arc Consistency (MAC) based on the AC-3 algorithm. The second algorithm implements a local search genetic algorithm (LSGA) inspired by recent research, treating potential Sudoku solutions as chromosomes within a population. This approach applies evolutionary processes like crossover, mutation, and local search to improve solution quality, with an elite learning mechanism to balance exploration and exploitation.

Beyond Sudoku, these algorithms have broader implications for CSPs and optimization challenges in real-world contexts, such as scheduling, resource allocation, and logistics. This work demonstrates how structured game problems like Sudoku can serve as experimental grounds for advancing AI methodologies applicable to complex problem-solving tasks.

Contents

List of Tables

List of Algorithms

Chapter 1

Introduction

Sudoku is a globally popular puzzle that involves filling a 9x9 grid with digits from 1 to 9, ensuring each row, column, and 3x3 subgrid contains **every digit exactly once** [1]. Despite its simple rules, Sudoku' s complexity scales rapidly with the number of empty cells, making it an intriguing problem for artificial intelligence (AI) [2]. This project explores two distinct AI-based approaches for solving Sudoku, comparing their efficiency and performance across a large dataset.

The first algorithm interprets the puzzle as Constraint Satisfaction Problem (CSP) and, thus, is based on the classical **Constraint Propagation** approach combined with **Backtracking**. This algorithm integrates advanced techniques such as forward checking and heuristics like **Minimum Remaining Values** (MRV) and **Least Constraining Value** (LCV), which guide the search by focusing on cells with the highest constraints or values that cause the least conflict [2]. The Maintaining Arc Consistency (MAC) component, based on the AC-3 algorithm, further enhances efficiency by minimizing the search space through domain reduction [2]. This structured approach allows the algorithm to systematically and efficiently prune impossible values, making it effective for puzzles with moderate to high numbers of clues.

The second algorithm employs a local search genetic algorithm (LSGA) based on a recent study in the literature [4]. Evolutionary approaches treat Sudoku solutions as "chromosomes" within a population of potential solutions and iteratively refines them through processes inspired by natural selection, such as crossover and mutation [2]. The LSGA also incorporates local search techniques, aiming to eliminate repeated numbers within each row, column, and subgrid by swapping values and selectively reinitializing cells [4]. Additionally, the study introduces an elite population learning mechanism, which stores the best solutions across generations, represents the usage of a global search. This helps balance the algorithm's exploration of new solutions with the refinement of promising ones. This approach explores an alternative pathway to solving structured puzzles like Sudoku, demonstrating the adaptability of evolutionary algorithms in constraint-driven contexts.

We utilize a dataset from **Kaggle**, containing four million Sudoku puzzles, each with a unique solution and varying levels of difficulty based on the number of pre-filled cells (clues), to evaluate and compare these two approaches. Key metrics include speed, memory usage and backtracking or iteration counts. Additionally, algorithm-specific metrics, such as constraint violations or chromosome fitness, are added to provide further insight into the efficiency of each approach under different levels of puzzle difficulty.

Chapter 2

Related Work

2.1 The Game of Sudoku

Sudoku is a logic-based number puzzle that has captivated players worldwide since it was popularized in the 1980s [1]. The goal of the game is to complete a 9x9 grid such that each cell contains a digit from 1 to 9, following three core rules:

- **Row Rule:** Each row in the 9×9 grid must contain every number from 1 to 9 exactly once. Therefore, every row must have nine unique values with no repetitions.
- **Column Rule:** Similarly, each column in the grid must contain every number from 1 to 9 exactly once. Like the row rule, this constraint requires all nine values in a column to be distinct.
- **Box Rule:** The grid is divided into nine 3×3 sub-grids or "boxes," and each of these boxes must also contain every number from 1 to 9 exactly once. The box rule imposes an additional layer of constraints, ensuring that no repetitions occur within each sub-grid.

Together, these rules impose constraints that make some solutions unique, adding both structure and complexity to the puzzle [5].

2.2 Constraint Satisfaction Problems and Constraint Propagation Algorithms

Constraint Satisfaction Problems (CSPs) are mathematical models that represent a class of problems defined by a set of variables, each associated with a domain of possible values, and a set of constraints that restrict the allowable combinations of values for these variables [2]. A CSP is considered solved when each variable is assigned a value from its domain in such a way that all constraints are satisfied. CSPs are a central topic in

artificial intelligence and are applicable to a wide variety of fields, including scheduling, resource allocation, and combinatorial puzzles such as Sudoku [2].
Constraint propagation algorithms play a significant role in solving CSPs efficiently. These algorithms reduce the domains of variables by enforcing constraints across variables iteratively, thereby narrowing down the search space before any assignment is finalized. One of the most widely used approaches in constraint propagation is **backtracking**, a recursive algorithm that incrementally builds solutions, attempting to assign values to variables in a way that satisfies all constraints. When a variable's assignment leads to a conflict with existing constraints, the algorithm *backtracks* to the previous variable assignment and tries a different value.
Enhanced versions of backtracking, such as **Forward Checking** and **Maintaining Arc Consistency (MAC)**, have been developed to further improve performance. Forward Checking preemptively reduces the domains of unassigned variables that share constraints with the current variable assignment, allowing early detection of conflicts [2]. MAC, based on the AC-3 (Arc Consistency 3) algorithm, goes even further by enforcing arc consistency throughout the search, ensuring that every variable pair meets the constraints [2]. These strategies, combined with heuristics like Most Constrained Variable (MCV) and Least Constraining Value (LCV), allow constraint propagation algorithms to solve complex CSPs with minimal backtracking, making them particularly effective for structured problems like Sudoku [2].

2.2.1 Sudoku as a Constraint Satisfaction Problem

Sudoku can be formulated as a **Constraint Satisfaction Problem (CSP)**, a problem type where each state is represented by a set of variables, each assigned a value from a domain. A CSP is solved when each variable is assigned a value that satisfies all imposed constraints, where these constraints are defined by the problem's rules.
Formally, a CSP is defined by three main components: X, D, and C [2]:

- **Variable Set** X: The set of variables, represented as $X = \{x_1, \ldots, x_n\}$.
- **Domain** D: Each variable x_i has an associated domain set $D = \{d_1, \ldots, d_n\}$, which contains all possible values the variable can acquire.
- **Constraints** C: A set of constraints or limits that defines the possible assignments for the values in D, ensuring compatibility between variables.

In the context of Sudoku, we can interpret these components in the following way:

- X: The set of all 81 cells in the grid.
- D: Each cell's domain D is the set of numbers from 1 to 9.

- C: The constraints that define valid assignments for each cell. Sudoku constraints can be divided into two types:

  - **Direct Constraints:**

    * **Row Constraint:** Each value within the same row must be unique.
    * **Column Constraint:** Each value within the same column must be unique.
    * **Box Constraint:** Each value within the same 3×3 sub-grid (or box) must be unique.

  - **Indirect Constraint:** Each number in the range 1 to 9 must appear exactly once per row, column, and box. These constraints further restrict the domain of each cell based on the already filled cells, thus limiting possible values for unfilled cells.

2.3 Genetic Algorithms: Concept and Applications

Genetic Algorithms (GAs) are a class of optimization and search algorithms inspired by the principles of natural selection and genetic evolution [3]. Originating from the field of evolutionary computation, GAs emulate the process of biological evolution to explore complex search spaces and optimize solutions to challenging problems. The core idea behind GAs is to represent potential solutions to a problem as *chromosomes* and evolve a population of these solutions over successive generations [2]. Through genetic operations such as **selection**, **crossover** (recombination), and **mutation**, GAs iteratively refine the population, enabling the fittest solutions to emerge over time. The selection process promotes solutions with higher fitness scores, ensuring that desirable traits are preserved in the next generation, while crossover and mutation introduce diversity, helping to explore new regions of the search space.

Genetic Algorithms are particularly useful for tackling optimization problems where the solution landscape is vast, multimodal, or difficult to navigate using traditional methods. They have been successfully applied in fields such as scheduling, engineering design, machine learning, and combinatorial optimization. GAs are also well-suited to problems that lack explicit mathematical formulations, as they only require a fitness function to evaluate solution quality.

Sudoku, though a seemingly simple puzzle, presents complex combinatorial challenges due to its inherent constraints and large search space. Genetic Algorithms (GAs), inspired by the process of natural selection, offer a compelling approach to solve such puzzles by iteratively evolving a population of candidate solutions.

2.3.1 Encoding Sudoku as a Genetic Algorithm Problem

In the context of GAs, Sudoku can be interpreted as a **Constraint Optimization Problem**, where the primary goal is to find a configuration of numbers that satisfies the puzzle's constraints without conflicts. The following components define the Sudoku problem within the GA framework:

- **Chromosomes (Individuals):** Each candidate solution to the Sudoku puzzle, or *individual*, is represented by a 9×9 grid filled with numbers from 1 to 9. Within each individual, rows, columns, and 3×3 boxes correspond to genes, which encode values that must collectively satisfy the constraints.

- **Population:** The GA maintains a population of candidate Sudoku grids, where each grid represents a possible solution state. The evolutionary process iteratively modifies this population, selecting and transforming individuals over successive generations to optimize their fitness in meeting the Sudoku constraints.

- **Fitness Function:** The fitness function in Sudoku GAs quantifies the number of constraint violations within an individual, counting repeated values within rows, columns, and boxes. The fitness score directly indicates the quality of each solution, with an optimal score achieved when all constraints are satisfied. This function serves as the primary objective measure, guiding the selection process towards increasingly accurate configurations.

Chapter 3

Methodology

This chapter elaborates on methods employed in this research, starting from data collection down to algorithm evaluation. Understanding these approaches is a mandatory pre-requirement, as it allows one to assess how well state-of-the-art models perform.

3.1 Dataset Overview: 4 Million Sudoku Puzzles

The dataset used for evaluating the Sudoku-solving algorithms in this project was created by RyanAn and is publicly available on Kaggle[1]. This dataset is specifically designed to test the efficiency and effectiveness of Sudoku algorithms across a wide spectrum of puzzle difficulties, ranging from easy to extremely challenging levels. The primary indicator of a puzzle's difficulty level is the number of **clues** (pre-filled digits) provided in the initial Sudoku grid, where puzzles with more clues tend to be easier, while those with fewer clues are more complex.

3.1.1 Dataset Characteristics and Structure

The dataset contains a total of **4 million unique Sudoku puzzles**, each paired with a solution to allow for verification and assessment of the algorithm's accuracy. The puzzles are distributed across various levels of difficulty, allowing for a robust evaluation of algorithm performance under different levels of constraint. Key characteristics of the dataset are detailed below:

- **Difficulty and Clues:**
  - Each puzzle contains between **17 and 80 clues**.
  - **Easy puzzles**, with up to 80 clues, can be solved more quickly as they provide ample initial information.

[1] `https://www.kaggle.com/ryanaherman/sudoku`

  - **Hard puzzles** contain as few as 17 clues, which is generally considered the minimum necessary to ensure a logically solvable puzzle.
  - The **magic number of 17 clues** is significant, as it represents the lowest clue count at which a Sudoku puzzle can maintain a single, logic-based solution.

- **Puzzle Distribution:** The dataset includes **62,500 puzzles for each clue count** from 17 to 80, thus ensuring a comprehensive and balanced coverage of various difficulty levels.

- **Dataset Format:** Each entry in the dataset consists of:
  - A **puzzle** string representing the initial Sudoku grid with 0s indicating empty cells.
  - A **solution** string that represents the fully completed Sudoku grid.
  - A **clue number** integer that represents the number of non-empty cells.

- **Uniqueness of Solutions:** Due to the mathematical structure of Sudoku, certain puzzles might exhibit multiple valid solutions. Users are encouraged to compare their computed solutions with those provided in the dataset, particularly when processing large batches of puzzles, to verify the algorithm's consistency and accuracy.

3.1.2 Application in this Project

In this project, we evaluate our algorithms on **five random samples of 50 puzzles each**, drawn from the dataset. Each sample is curated to represent increasing levels of difficulty, with two samples specifically constrained to a strict clue range (less than 25 clues) to test the algorithms' robustness on very challenging problems. This approach allows us to observe the behavior of each algorithm across a gradient of difficulty, providing insight into performance trends and potential bottlenecks.
The dataset serves as a comprehensive benchmark for assessing both the efficiency and accuracy of Sudoku-solving algorithms in a controlled, scalable manner.

3.2 Constraint Propagation-Based Sudoku Solver with Backtracking

In this section, we describe the implementation of a constraint propagation-based algorithm for solving Sudoku puzzles, leveraging advanced techniques in constraint satisfaction and heuristic search. This algorithm, implemented as the `SudokuSolverMAC` class, combines constraint propagation with backtracking and incorporates several optimizations, including Maintaining Arc Consistency (MAC), Most Constrained Variable (MRV), and Least Constraining Value (LCV) heuristics.

3.2.1 Overview of the Approach

The `SudokuSolverMAC` algorithm treats Sudoku as a Constraint Satisfaction Problem (CSP), where each cell in the grid is considered a variable constrained by neighboring cells in the same row, column, and 3x3 subgrid. In a CSP, the objective is to find values for each variable that satisfy all given constraints.
The algorithm proceeds in three main stages:

1. **Initialization of Domains**: Sets an initial domain of possible values for each cell, based on the input puzzle.

2. **Constraint Propagation**: Prunes possible values in each cell's domain by enforcing arc consistency and applying initial constraints.

3. **Backtracking with Heuristics**: Uses MRV and LCV heuristics to optimize the search for a valid solution if constraint propagation alone cannot solve the puzzle.

3.2.2 Implementation Details

The `SudokuSolverMAC` class is designed to process a standard 9x9 Sudoku grid, where empty cells are represented by `0`. The primary components of the class and their functions are described below.

`initialize_domains`

The `initialize_domains` method defines the initial domain for each cell based on the input grid. Cells that contain a number from the start have their domains limited to that single value, while empty cells are initialized with a full domain of $\{1, 2, \ldots, 9\}$. This setup provides a baseline for further constraint propagation.

`update_domains`

The `update_domains` method performs initial constraint propagation by updating the domains of neighboring cells when a value is placed in a cell. This ensures that:

- No other cell in the same row, column, or subgrid can contain the placed number.

- The domains of neighboring cells are narrowed down based on this constraint, simplifying the puzzle and reducing possible conflicts.

`preprocess_domains`

This function applies *forward checking* by limiting possible values in each cell's domain based on the values in its neighbors. If a cell's domain is restricted to one value, this

method propagates the constraint to the neighboring cells in the same row, column, or subgrid, further reducing the domains and preparing the grid for backtracking.

get_neighbors

The `get_neighbors` function identifies the set of all neighboring cells in the same row, column, or subgrid. This set of neighbors is used by the `update_domains` and `revise` functions to propagate constraints efficiently.

revise

The `revise` function is central to maintaining arc consistency. Given two cells (variables) `xi` and `xj`, this function ensures that every value in `xi`'s domain is consistent with `xj`'s domain. If `xj` has only one possible value, that value is removed from `xi`'s domain if present, helping to propagate constraints and eliminate potential conflicts.

maintain_arc_consistency

This method enforces global arc consistency across the puzzle. It initializes a queue of all arcs (pairs of neighboring cells) and uses the `revise` function to check for inconsistencies, removing values from domains as necessary. If a cell's domain is revised, its neighbors are re-added to the queue to propagate the changes, ensuring consistency throughout the grid.

find_most_constrained_cell

The `find_most_constrained_cell` method applies the Minimum Remaining Values (MRV) heuristic to find the cell with the fewest remaining values in its domain. This cell is likely to be the most constrained and prone to conflicts, making it a suitable candidate for early exploration in the search process.

get_least_constraining_values

This function implements the Least Constraining Value (LCV) heuristic, which prioritizes values that rule out the fewest options for neighboring cells. This approach increases the likelihood of maintaining flexibility in the solution space, thus reducing the need for backtracking.

solve

The `solve` function integrates all components to recursively explore possible solutions using backtracking. The MRV and LCV heuristics guide the choice of cells and values, while arc consistency and forward checking reduce the search space, making the algorithm

more efficient. This function assigns values to cells, updates domains, and backtracks if necessary, ultimately returning a solution if one exists.

Algorithm 1 Sudoku Solver with Constraint Propagation and Backtracking

```
1: procedure SOLVE
2:     PREPROCESS_DOMAINS                                   ▷ Initial constraint propagation
3:     (row, col) ← FIND_MOST_CONSTRAINED_CELL              ▷ MRV heuristic
4:     if row = None and col = None then
5:         return True                                      ▷ Puzzle solved if no empty cell remains
6:     end if
7:     possible_values ← GET_LEAST_CONSTRAINING_VALUES(row, col)   ▷ LCV heuristic
8:     for each value in possible_values do
9:         grid[row][col] ← value                           ▷ Assign the value
10:        original_domains ← copy of domains               ▷ Backup current state
11:        domains[row][col] ← {value}
12:        if MAINTAIN_ARC_CONSISTENCY then
13:            if SOLVE then
14:                return True                              ▷ Continue to solve recursively
15:            end if
16:        end if
17:        grid[row][col] ← empty_cell                      ▷ Backtrack: revert assignment
18:        domains ← original_domains                       ▷ Restore previous domains
19:    end for
20:    return False                                         ▷ No solution found, backtracking required
21: end procedure
```

3.3 Local Search Genetic Algorithm for Sudoku

The `SudokuLSGA` class implements a recently developed genetic algorithm (GA) approach tailored to solving Sudoku puzzles, following the principles outlined in the paper *A Novel Evolutionary Algorithm With Column and Sub-Block Local Search for Sudoku Puzzles* [4]. In this framework, the Sudoku grid is represented as a population of **chromosomes**, each of which undergoes iterative improvement through evolutionary operations. This approach leverages both genetic operators (such as crossover and mutation) and problem-specific local search techniques (column and sub-block local search) to explore the solution space efficiently.

3.3.1 Representing Sudoku as a Genetic Algorithm

Under the presented genetic algorithm, a candidate solution (individual) is often represented as a *chromosome* composed of *genes*, which collectively encode the solution in a structured format. For the Sudoku problem:

- Each **chromosome** represents a complete 9x9 Sudoku grid, where empty cells in the grid are assigned values that satisfy row constraints.
- Each **gene** represents a cell in the Sudoku grid, and the value of each gene falls within the set $\{1, 2, \ldots, 9\}$.

Thus, the objective is to evolve a population of Sudoku grids (chromosomes) until reaching a state where all rows, columns, and sub-blocks of each grid contain the digits from 1 to 9 exactly once.
In this work the single individual is represented by two matrices:

- The **major matrix** is the matrix that is specific to each individual as it represents the chromosome itself. This means every major matrix represents the evolution of the grid across the generations for an individual. It is directly paired with the fitness of the individual, to keep track of the improvement during the phases.
- The **associated matrix** is a 2D 9x9 matrix which keeps track of what numbers are provided from the initial puzzle and which cells are empty. In our case, the elements of the matrix with the value 1 are assigned numbers. this matrix never changes as it is needed for re-initialization purposes. Thus, we decided to share it across the whole process and cannot be updated even if numbers are assigned to the corresponding individuals.

3.3.2 Algorithm Overview

The `SudokuLSGA` class operates within an evolutionary framework, iterating through steps that include:

1. **Population Initialization**: An initial population of Sudoku grids is created by assigning random values to empty cells.

2. **Selection and Reproduction**: Individuals are selected based on fitness, guiding reproduction toward promising solutions.

3. **Crossover and Mutation**: Crossover exchanges rows between parents, while mutation introduces diversity by swapping or reinitializing values.

4. **Local Search**: Column and sub-block local search refine candidate solutions, reducing conflicts within columns and sub-blocks.

5. **Elite Population Learning**: The algorithm maintains an elite set of the best solutions to prevent stagnation and improve convergence. This elitism is used to replace the worst individuals for each generation with a random elite individual or the complete reinitialization.

Each of these methods contributes uniquely to optimizing the population towards a valid Sudoku solution.

3.3.3 Implementation Details

This section provides a detailed breakdown of each method within the `SudokuLSGA` class, discussing their roles, contributions, and implementation.

GA Configuration - `GAConfig`

The `GAConfig` class contains essential parameters that define the genetic algorithm's behavior:

- **Population Size** controls the number of individuals (Sudoku grids) in each generation. A larger population increases diversity, reducing the risk of premature convergence.

- **Elite Size** determines how many top-performing individuals are preserved in each generation, providing a way to guide the population towards high-quality solutions.

- **Crossover and Mutation Rates** are 4 parameters that control the probabilities of crossover and mutation events. These rates balance exploration and exploitation in the search space.

Population Initialization - `initialize_population`

The `initialize_population` method constructs an initial population of candidate Sudoku solutions. Since the class stores the initial grid and the associated matrix, each individual in the population is created through a random initialization of the empty cell to values that respect the row constraint of Sudoku (i.e., each row contains unique numbers from 1 to 9):

- **Row Constraint Satisfaction**: By ensuring each row contains unique values initially, the algorithm reduces the complexity of the search space, since rows only need local refinement to address column and sub-block constraints.
- **Population Diversity**: This method generates multiple randomized versions of the Sudoku grid, contributing to a diverse initial population that supports a broad exploration of potential solutions.

Fitness Evaluation - `fitness` and `evaluate_population`

The `fitness` function measures the quality of each individual based on the Sudoku rules for columns and sub-blocks. The total fitness value of an individual is computed as sum of two measures:

- **Column Fitness**: For each individual, the number of illegal columns (i.e., columns containing duplicate values) is counted.
- **Sub-Block Fitness**: For each 3x3 sub-block, the number of duplicate values is recorded.

The sum of these conflicts defines the fitness score, with lower scores indicating fewer violations. By focusing on column and sub-block conflicts, this function leverages the structure of the initialized individuals, which already satisfy row constraints, making the search more efficient.
The `evaluate_population` method iterates through the entire population, assigning a fitness score to each individual, enabling effective selection and reproduction based on solution quality.

Selection - `tournament_selection`

The `tournament_selection` method selects individuals for reproduction using a tournament selection strategy:

- **Tournament Mechanism**: For each selection, two individuals are chosen randomly, and the individual with the lower fitness (fewer conflicts) is selected.

- **Balancing Exploration and Exploitation**: This selection method ensures that fitter individuals are more likely to contribute to the next generation, thus guiding the population toward higher-quality solutions without completely discarding less fit individuals, which might contain valuable diversity.

Crossover - `crossover`

The crossover function facilitates recombination of selected parents to generate offspring, preserving the row uniqueness within Sudoku constraints.

- **Parent Selection**: Pairs of parents are selected for crossover based on an individual crossover rate (PC1).
- **Row-Based Crossover**: Each row has a row crossover rate (PC2), determining whether it is swapped between the parents. This process introduces diversity in offspring while respecting the row constraint, preserving the structure established during initialization.

By allowing only row-based exchanges, this function maintains the row constraints of the Sudoku puzzle, reducing the solution space while providing diverse combinations.

Mutation - `mutate`

The `mutate` function introduces randomness to avoid premature convergence by modifying individuals within the population:

- **Swap Mutation (PM1)**: Randomly selects two positions in a row and swaps their values. This mutation maintains row constraints, allowing the algorithm to explore configurations that might not emerge from crossover alone.
- **Reinitialization Mutation (PM2)**: Replaces an entire row with a new random permutation of values from 1 to 9. This technique enhances exploration, especially in scenarios where a local search fails to yield improvements.

These mutation techniques allow the algorithm to explore new configurations while ensuring that row constraints remain satisfied.

Column and Sub-Block Local Search

Local search functions enhance solution quality by refining individual columns and sub-blocks to remove duplicates:

- **Column Local Search**: Identifies and corrects columns containing repeated values by swapping them with values from other columns within the same row. This function focuses on resolving conflicts while retaining row constraints.

- **Sub-Block Local Search**: Similar to column local search, but applied to 3x3 sub-blocks. This process eliminates duplicates within sub-blocks, using swaps with non-repeating values in adjacent sub-blocks.

By addressing conflicts within columns and sub-blocks, these local search operations significantly improve fitness without altering the row constraints established during initialization.

Elite Population Learning - `elite_population_learning`

The elite learning strategy helps the algorithm avoid stagnation by preserving the best individuals from previous generations:

- **Replacement Strategy**: The worst-performing individual in the current population is replaced with an elite individual based on a probability calculated from the fitness difference. This introduces a balance between exploiting high-quality solutions and exploring new search directions.
- **Historical Learning**: By retaining the best solutions across generations, the algorithm can leverage previous discoveries, improving the overall quality of solutions in the population.

This mechanism provides a memory of high-quality solutions, guiding the population toward convergence without losing valuable configurations.

3.3.4 Algorithm Workflow - `evolve`

The `evolve` function represents the main iterative loop of the genetic algorithm:

1. **Population Initialization**: Initializes a population of individuals, each representing a potential solution to the Sudoku puzzle.
2. **Evaluate Fitness**: Computes the fitness scores for each individual, assessing the number of conflicts within columns and sub-blocks.
3. **Iterative Evolution**: Executes selection, crossover, mutation, local search, and elite learning over multiple generations until an optimal solution is found or the maximum generation count is reached.
4. **Termination**: The loop exits when an individual with zero fitness (a valid Sudoku solution) is found or when the maximum number of generations is reached.

This iterative process allows the algorithm to refine individuals over generations, gradually reducing conflicts and optimizing fitness scores.
Here we present a rough pseudo-code of the evolve method:

Algorithm 2 Sudoku Solver using Local Search Genetic Algorithm

```
1: procedure EVOLVE
2:     count ← 0                                          ▷ Track the number of generations
3:     best_individual ← None   ▷ Tuple used to track the best individual found and its fitness
4:     population ← INITIALIZE_POPULATION    ▷ Initialize population and their fitness
5:     population ← EVALUATE_POPULATION(population)   ▷ Updates the fitness of the population
6:     elite ← empty              ▷ Create the MIN-PQ like structure for elite individuals
7:     while count < max_generations do
8:         population ← TOURNAMENT_SELECTION(population)   ▷ Perform tournament selection
9:         population ← CROSSOVER(population)          ▷ Perform crossover to create offspring from parents
10:        population ← MUTATE(population) ▷ Perform mutation within individuals or re-initialization
11:        population ← COLUMN_LOCAL_SEARCH(population)          ▷ Swap of values between columns within individuals
12:        population ← SUBBLOCK_LOCAL_SEARCH(population)          ▷ Swap of values between subgrids within individuals
13:        population ← EVALUATE_POPULATION(population)    ▷ Update the fitness of the modified population
14:        elite ← UPDATE_ELITE_POPULATION(population, elite)          ▷ Update elite individuals queue
15:        population ← ELITE_POPULATION_LEARNING(population, elite)   ▷ Substitute with random elite or reinitialize worse individual in generation
16:        best_individual ← elite.copy_min              ▷ Store the best individual found
17:        if best_fitness == 0 then
18:            return best_individual                    ▷ If the solution is optimal, return
19:        end if
20:        count ← count + 1                             ▷ Increment the generation count
21:    end while
22:    return best_individual                            ▷ Return the best found solution
23: end procedure
```

3.4 Metrics and Benchmarking Measures

To provide an appropriate benchmarking process, particularly for solving computationally intensive problems such as Sudoku, several key performance metrics are essential for a thorough evaluation of algorithm efficacy. These metrics include **execution time**, **CPU usage**, and **memory consumption**, each providing distinct insights into the algorithm's behavior under different conditions. We successfully integrated these metrics into our evaluation framework by utilizing Python libraries such as `psutil` for resource monitoring and `threading` for concurrent execution.

To capture these metrics concurrently with the solver's operations, we implemented a `threading` approach. A separate thread continuously samples CPU and memory usage at regular intervals while the main solver thread executes the puzzle-solving algorithm. This design allows for real-time monitoring without introducing significant overhead or affecting the algorithm's execution flow. The continuous sampling ensures that we collect multiple data points for each metric, allowing us to compute statistics such as minimum, average, and maximum values after each solving iteration.

Furthermore, we capture **solver-specific statistics** that provide deeper insights into each algorithm's inner workings. For LSGA, metrics such as the number of generations, fitness evaluations, and mutation rates are recorded to assess the effectiveness of the genetic operations in finding solutions. For MAC, we track the number of constraint checks and propagations, which highlight the algorithm's ability to maintain arc consistency and reduce search space efficiently.

The results from this comprehensive evaluation can be found inside the jupyter notebook used for the analysis.

Chapter 4

Experiments and Results

In this chapter, the study provides a comparison of the algorithms, focusing on their performance on random samples. Additionally, some critique and ambiguous aspects from this methods are clearly mentioned, so that future work can workaround this issue. This comparison offers a clear view of how these models perform under similar conditions and what challenges remain unaddressed.

4.1 Experimental Setup

Component	**Description**
Operating System (OS)	Microsoft Windows 10 Pro, version 22H2, build 19045.5011 (64-bit)
Motherboard	MSI B450 GAMING PRO CARBON AC (MS-7B85)
Central Processing Unit (CPU)	AMD Ryzen 7 5800X, 8-Core Processor, 3.8 GHz, 8 Cores, 16 Logical Processors
Memory (RAM)	4 x 8 GB Corsair, 3200 MHz (Total: 32.0 GB)
Graphics Processing Unit (GPU)	Nvidia RTX 2080 Super
Solid State Drive (SSD)	1 TB
Hard Disk Drive (HDD)	2 TB

Table 4.1: Computer Hardware and Software Specifications

4.2 Benchmark Results

Please refer to the jupyter notebook inside the project for further information about the benchmark results.

Chapter 5

Limitations and Discussion

In this section, it is proposed an examination of the key limitations identified during the evaluation of the approaches. Despite notable advancements, these algorithms face real challenges when they try to maintain real-time performance under sparse clues scenarios. This analysis invites the read to a critical reflection on where these models fall short and suggests areas where further research can flourish.

5.1 Pros and Cons of the MAC-based Constraint Propagation Algorithm with Backtracking for Sudoku Solving

5.1.1 Consistency Enforcement

- **Pros**: The MAC algorithm is highly effective at enforcing arc consistency, **significantly reducing the search space** by pruning values from variable domains that are incompatible with constraints. This is especially useful in Sudoku, where eliminating impossible values early accelerates the solving process.

- **Cons**: While arc consistency reduces search space, it **does not guarantee a unique solution** or full constraint satisfaction. The algorithm may need to backtrack if the current path leads to a dead end, requiring additional **computational overhead** for constraint re-evaluation. Plus the current implementation does not use a very developed backtracking technique to balance this aspect.

5.1.2 Efficiency in Pruning

- **Pros**: MAC reduces unnecessary exploration of invalid solutions by maintaining arc consistency dynamically throughout the search. This proactive pruning **limits**

backtracking, helping the algorithm focus only on feasible solution paths, which improves overall efficiency.

- **Cons**: Enforcing arc consistency is computationally expensive. The MAC algorithm must continually revise domains and process arc consistency checks, which can **slow down performance**, particularly in early stages where domains are larger.

5.1.3 Domain Reduction (Forward Checking)

- **Pros**: MAC employs forward checking, which reduces domains by removing values from neighboring cells when assignments are made. This targeted approach **helps eliminate infeasible paths early**, improving the likelihood of finding solutions without excessive backtracking.

- **Cons**: In cases where constraints are complex or highly interdependent, forward checking alone may be insufficient. Without full look-ahead, forward checking can **miss deeper constraint violations**, necessitating additional backtracking and causing inefficiencies in complex puzzles.

5.1.4 Backtracking Reduction

- **Pros**: MAC significantly reduces the need for extensive backtracking by maintaining domain consistency and applying heuristics such as Most Constrained Variable (MRV) and Least Constraining Value (LCV). This is particularly advantageous for problems like Sudoku, which benefit from systematic pruning.

- **Cons**: Although reduced, backtracking can still be necessary, especially in high-complexity Sudoku puzzles with sparse clues. If the algorithm reaches a path that requires substantial undoing of prior work, the efficiency gains from backtracking reduction can be negated by high computational costs.

5.1.5 Heuristic Integration

- **Pros**: MAC's performance is enhanced by heuristics, including MRV (selecting the most constrained cell) and LCV (choosing values that impose the least constraints on neighbors). These heuristics guide the search toward promising paths and reduce unnecessary branching.

- **Cons**: The choice of heuristics can influence MAC's effectiveness. Relying too heavily on one heuristic (e.g., MRV) may lead to premature selections that force the algorithm to backtrack. Balancing heuristics is crucial, as poor choices can misguide the search, especially in tightly constrained grids.

5.2 Pros and Cons of the Local Search Genetic Algorithm (LSGA) for Sudoku Solving

5.2.1 Search Efficiency

- **Pros**: LSGA leverages local search techniques (column and sub-block) to explore the solution space more effectively, quickly reducing the number of constraint violations in columns and sub-blocks. This targeted approach allows it to **escape simple local optima** and move toward promising areas of the search space.

- **Cons**: Despite efficient exploration, LSGA can still **struggle with complex local optima**, particularly when initial populations are far from feasible solutions. Achieving a truly optimal solution can be slow, especially for higher-difficulty Sudoku puzzles where sub-block constraints are tightly interwoven.

5.2.2 Exploration and Exploitation Balance

- **Pros**: By combining traditional genetic algorithm operators (mutation and crossover) with local search adjustments, LSGA **balances exploration** (searching new areas) with exploitation (refining current solutions). This adaptive balance is beneficial for Sudoku solving, as both broad search and detailed refinement are needed.

- **Cons**: This approach may still **lean too heavily on exploitation**, especially in later stages, leading to premature convergence if population diversity is not adequately maintained. In complex instances, there is a risk of the algorithm being trapped in specific suboptimal configurations.

5.2.3 Fitness Evaluation

- **Pros**: The fitness function in LSGA is specifically designed to count constraint violations across columns and sub-blocks, making it effective in guiding the algorithm toward valid Sudoku solutions. This **tailored fitness metric** helps the algorithm to gradually approximate a solution by minimizing errors step-by-step.

- **Cons**: Fitness **evaluations are computationally intensive** as they require evaluating each column and sub-block constraint for every individual. For large populations and repeated generations, this evaluation can be resource-intensive, impacting overall efficiency in scenarios with high processing constraints.

5.2.4 Elitism and Population Learning

- **Pros**: LSGA uses an elite population learning mechanism to incorporate the best solutions from previous generations. This historical learning prevents the algorithm from discarding valuable high-quality solutions, **promoting convergence toward optimal** solutions over generations.
- **Cons**: Retaining elite individuals can lead to **reduced diversity** in the population, particularly if elite solutions dominate successive generations. If not balanced with enough reinitialization or exploration, this elitism may cause stagnation in the evolutionary process.

5.2.5 Parameter Dependence

- **Pros**: LSGA's performance is **highly tunable**, allowing practitioners to adjust population size, mutation rates, crossover rates, and elite sizes to optimize performance based on Sudoku difficulty. This adaptability is an advantage in tailoring the algorithm to different problem instances.
- **Cons**: Performance is **sensitive to parameter settings**, and finding optimal values for these parameters can require significant experimentation. Over-tuning for specific Sudoku instances may make the algorithm less generalizable and effective on puzzles with varying structures and difficulty levels.

5.3 Conclusive Considerations

While both algorithms bring unique strengths, they also face limitations that can be addressed by integrating additional techniques. LSGA's heuristic-driven approach may benefit from hybridizing with MAC's constraint propagation for greater precision in identifying valid solution candidates, or by adding a post-processing phase to check and correct any constraint violations. MAC's performance could be improved in larger puzzles by incorporating probabilistic methods or randomized restart strategies to mitigate potential bottlenecks from extensive backtracking. Ultimately, LSGA is best suited for large-scale exploratory problem-solving, while MAC is advantageous when the primary objective is to find exact, rule-compliant solutions. An ideal approach might combine elements of both algorithms to balance the speed and flexibility of LSGA with the exactness of MAC, enabling enhanced adaptability and precision across a variety of Sudoku-solving contexts.

Chapter 6

Conclusions

The `SudokuSolverMAC` class illustrates a structured, heuristic-driven approach to solving Sudoku, leveraging arc consistency and domain reduction techniques commonly used in CSPs. This implementation not only serves as a powerful Sudoku-solving tool but also provides insights into solving complex, constraint-driven problems in various real-world contexts. However, this implementation lacks of an intelligent approach to deal with Backtracking, such as using a Conflict-Directed-Backjump approach. In fact, the inference is sometimes too fast and leads to a wrong solution when the count of clues is small. The `SudokuLSGA` class demonstrates the effectiveness of genetic algorithms combined with local search techniques in solving structured constraint satisfaction problems (CSPs). While this approach is spatially expensive, it still achieves better accuracy when it comes to solving the puzzles. In fact, genetic algorithms are inherently resource-intensive, particularly for large populations and high mutation rates, but this can still be balanced by the use of fast libraries such as NumPy (or even the use of GPU supported libraries like PyTorch or Tensorflow). The algorithm may also face limitations with excessively complex Sudoku puzzles, requiring substantial computational resources to reach a valid solution. On the other hand, tweaking the parameters may just reduce this issue at the cost of introducing some bias in the inference process. In conclusion, both approaches show an impressive capability of tackling hard Sudoku puzzles and present plenty of space for improvement in both performance and accuracy.

Bibliography

[1] Delahaye Jean-Paul. The science behind sudoku. *Scientific American*, 294:80–7, 07 2006. doi: 10.1038/scientificamerican0606-80.

[2] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, Global Edition*. Pearson Education, 2021. ISBN 9781292401171. URL https://books.google.it/books?id=cb0qEAAAQBAJ.

[3] Mantere Timo and Koljonen Janne. Solving, rating and generating sudoku puzzles with ga. In *2007 IEEE Congress on Evolutionary Computation*, pages 1382–1389, 2007. doi: 10.1109/CEC.2007.4424632.

[4] Chuan Wang, Bing Sun, Ke-Jing Du, Jian-Yu Li, Zhi-Hui Zhan, Sang-Woon Jeon, Hua Wang, and Jun Zhang. A novel evolutionary algorithm with column and sub-block local search for sudoku puzzles. *IEEE Transactions on Games*, 16(1):162–172, 2024. doi: 10.1109/TG.2023.3236490.

[5] Robin Wilson. Sudoku. *Encyclopedia Britannica*, July 2024. URL https://www.britannica.com/topic/sudoku. Accessed 27 October 2024.