Università
Ca'Foscari
Venezia

Artificial Intelligence and Data Engineering

Foundation of Artificial Intelligence

Assignment 1: **Sudoku Solver**

Professor
Andrea Torsello

Autor
Davide Tonetto
Student ID 884585

Academic year
2023/2024

Contents

List of Figures

Chapter 1

Introduction

1.1 Sudoku puzzle

Sudoku is a logic-based number placement game that requires you to fill a $9x9$ grid with digits so that each column, each row, and each of the nine $3x3$ sub-grids contain all the digits from 1 to 9 without any repetition. The game starts with some of the cells already filled in, and the objective is to fill the missing cells with the correct digit.

More precisely, the constraints to be satisfied are:

- **Cell constraint**: Each cell in the Sudoku grid must contain a digit from 1 to 9.
- **Row constraint**: Each of the 9 rows in the grid must contain all the digits from 1 to 9 with no repeating digits.
- **Column constraint**: Each of the 9 columns in the grid must contain all the digits from 1 to 9 with no repeating digits.
- **Sub-grid constraint**: Each $3x3$ sub-grid in the grid must contain all the digits from 1 to 9 with no repeating digits.

Formally, we can define the Sudoku puzzle as a $9x9$ matrix

$$A = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{08} \\ a_{10} & a_{11} & \cdots & a_{18} \\ \vdots & \vdots & \ddots & \vdots \\ a_{80} & a_{81} & \cdots & a_{88} \end{bmatrix}$$

with the following constraints:

- **Cell constraint**:

$$a_{i,j} \in [0, 8] : \forall\ a_{i,j} \in A$$

- **Row constraint**:

$$a_{i,j} \neq b_{i,k} : (a_{i,j}, b_{i,k} \in A)(\forall i, j, k \in [0, 8] \text{ s.t. } j \neq k)$$

- **Column constraint**:

$$a_{i,j} \neq b_{k,j} : (a_{i,j}, b_{k,j} \in A)(\forall i, j, k \in [0, 8] \text{ s.t. } i \neq k)$$

- **Sub-grid constraint**:

$$a_{p\cdot 3+i, q\cdot 3+j} \neq b_{p\cdot 3+k, q\cdot 3+w} : (a, b \in A)(\forall p, q, i, j, k, w \in [0, 3] \text{ s.t. } i \neq k \wedge j \neq w)$$

These constraints ensure that each number appears exactly once in each row, column, and sub-grid.

1.1.1 Example

Figure 1.1 shows an example of a Sudoku puzzle (a) and a possible solution for it (b).

						3		
		1	2		4		8	
								4
3		6						7
		7					6	
				3		8	2	
					1		4	
							9	8
		8	5			7		1

(a) Unsolved

7	8	4	1	6	9	3	5	2
5	3	1	2	7	4	6	8	9
6	9	2	3	5	8	1	7	4
3	2	6	8	9	5	4	1	7
8	5	7	4	1	2	9	6	3
4	1	9	6	3	7	8	2	5
2	7	3	9	8	1	5	4	6
1	6	5	7	4	3	2	9	8
9	4	8	5	2	6	7	3	1

(b) solved

Figure 1.1: Sudoku puzzle example

1.2 Purpose

This text proposes two different algorithms that solve a Sudoku puzzle given as input. One uses a constraint satisfaction approach, and the other treats the Sudoku puzzle as an optimization problem.

1.2.1 Implementation

Both solvers have been implemented in Python 3.10 and included in a class named "Sudoku". The class takes as input in the constructor a matrix from

a text file (giving the path to the file) or a flattened matrix from a string. Empty squares in the matrix are represented by the standard symbol '.', while known squares should be represented by the corresponding digit (1,...,9). For example:

37.	5..	..6
...	36.	.12
...	.91	75.
...	154	.7.
..3	.7.	6..
.5.	638	...
.64	98.	...
59.	.26	...
2..	..5	.64

Alternatively, it can be flattened as follows and stored in a string:

37.5....6...36..12....9175....154.7...3.7.6...5.638....6498....59..26...2....5.64

The solver class provides two methods for solving the puzzle and other utility methods used to print and verify the correctness of the solution. The methods are as follows:

- **is_correct()**: verifies if the current state of the Sudoku satisfies all the constraints (this method ignores the empty cells).
- **is_solved()**: checks if all the cells are not empty.
- **constraint_propagation_solver(...)**: the first solver.
- **genetic_solver(...)**: the second solver.

All these methods will be discussed in more detail in the next chapters.

The solution given by a solver is stored inside the Sudoku class and can be printed in the console using the print() function. For example:

```
sudoku = Sudoku(file_path) # initialize sudoku
# call solver 1
sudoku.constraint_propagation_and_backtracking_solver()

print(sudoku) # print solution
print(f'Solved: {sudoku.is_solved()}') # check if it is solved
print(f'Correct: {sudoku.is_correct()}') #check if it is correct
```

Lastly, to execute the code, it is necessary to install some packages. To do this, run "pip install" in the code's directory.

To run experiments is necessary to download the CSV file from the following link (3 million Sudoku puzzles with ratings) and add it to the tests folder.

Chapter 2

Constraint satisfaction approach

2.1 Defining Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) is a mathematical problem-solving framework used to represent and solve a wide range of combinatorial problems. A CSP consists of three components, X, D and C, where:

- X is a set of variables, $X_1, ..., X_n$. These represent the unknowns in the problem.
- D is a set of domains, $D_1, ..., D_n$. Domains represent the set of values that each variable can take. For example, in a Sudoku puzzle, each cell can have a domain of numbers from 1 to 9.
- C is a set of constraints that specify allowable combinations of values. Constraints can be unary (involving a single variable) or binary (involving two variables) and can also involve more than two variables.

The goal of a CSP is to find a consistent assignment of values to the variables such that all constraints are satisfied. In other words, the solution to a CSP is a set of values for the variables such that no constraint is violated.

CSP search algorithms take advantage of the structure of states and use general-purpose rather than problem-specific heuristics to enable the solution of complex problems. The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

2.1.1 Typologies of CSP

CSP problems can be categorized by domain and constraint type. We have two possible kinds of domain types:

- **Discrete domain**: In a discrete domain, the variable can take on values from a countable set, which means that there are a finite or countably infinite number of distinct values that the variable can assume. These values are typically separated by clear intervals. Therefore we have two sub-categories:
  - **Finite domain CSPs**: In these problems, variables have finite, discrete domains, meaning that the possible values for each variable

come from a well-defined, countable set. Examples include Sudoku puzzles and map colouring problems where you have a limited number of colours to choose from.

  - **Infinite domain CSPs**: In contrast to finite domain CSPs, variables in infinite domain CSPs can take values from an unbounded or continuous set. These are often encountered in scheduling and optimization problems, where time variables or real numbers may be involved.

- **Continuous domain**: In a continuous domain, the variable can take on values from an uncountable set, typically involving real numbers. This means there is an infinite number of possible values within a given range, and the values can be any real number within that range.

On the other hand, also constraints categorize CSP. We have the following types of constraints:

- **Unary constraints**: constraints that restrict the value of a single variable (simplest kind of constraints).
- **Bynary constraints**: constraints that involve only two variables at a time.
- **Global constraints**: constraints involving an arbitrary number of variables.

Understanding these typologies helps in choosing appropriate algorithms and techniques for solving CSPs, as the nature of the domain and constraints greatly influence the problem-solving approach.

In this text, we will focus on solving CSP with a discrete, finite domain (the category into which Sudoku falls).

2.2 Constraint propagation and Backtracking

In discrete, finite domain CSPs there is an important choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of inference called constraint propagation.

2.2.1 Constraint propagation

Constraint propagation is a technique used to simplify and solve Constraint Satisfaction Problems (CSPs). This technique uses constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on. The main idea is to reduce the domains of variables by removing values that cannot be part of a valid solution, which makes the search for a solution more efficient. The goal of constraint propagation is to enforce local consistency within the CSP. Local consistency

means that the constraints are satisfied for each constraint or group of related constraints. There are different types of local consistency, such as:

- **Node consistency**: A single variable is node-consistent if all the values in its domain satisfy the variable unary constraint. It is always possible to eliminate all the unary constraints in a CSP by running node consistency. It is also possible to transform all n-ary constraints into binary ones.
- **Arc consistency**: A variable is arc-consistent if every value in its domain satisfies the variable's binary constraints. More formally, X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j).

Unfortunately, not all CSP problems can be solved by inference alone, therefore is necessary to **search** for a solution. For this reason, in the next section, we will look at a backtracking search algorithm that works on partial assignments.

2.2.2 Backtracking search for CSPs

The term backtracking search is used for a depth-first search that explores the search space by trying different assignments of values to variables and undoing those assignments when they lead to dead-ends. It explores the entire solution space to find a solution if exists. Backtracking models the state space as a tree and explores it with a depth-first search.

Backtracking is combined with constraint propagation to solve CSPs. Constraint propagation helps to reduce the search space for the Backtracking search algorithm, making it more efficient. There are several common techniques used to apply constraint propagation in Backtracking search:

- **Arc Consistency**: infer reductions in the domain of variables *before* we begin the search by using AC-3 or other algorithms.
- **Forward Checking**: after assigning a value to a variable, check its neighbours' domains and remove values from them that are no longer consistent. Because forward checking only does arc consistency inferences, there is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

inference can be more powerful in the course of a search: every time we choose a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighbouring variables. For this reason, we choose Forward Checking for the Sudoku solver.

Variable and Value heuristics

To improve the backtracking search algorithm we can adopt some strategies that used to determine the order in which variables are selected and the values are assigned to those variables. The solver proposed in this text uses the following two heuristics:

- **Minimum Remaining Values (MRV)**: MRV heuristic selects the variable with the fewest remaining values in its domain. The idea is to focus on the variables that are most constrained and likely to cause problems if not handled early.
- **Least Constraining Value (LCV)**: LCV heuristic, often used in combination with MRV, chooses the value for a variable that rules out the fewest values in the domains of the remaining variables. This helps to maintain flexibility in future assignments.

We will see the benefits of using those heuristics in the following sections.

2.3 Sudoku puzzle as a CSP problem

We can easily represent a Sudoku puzzle as a CSP problem. We can model it in the following way:

- **Variables**: the variables of the CSP problem are represented by the cells of the Sudoku matrix (81 variables).
- **Domains**: for each variable its domain is represented by the set of number $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ that are all the possible values that can be assigned to a cell.
- **Constraints**: For each row, column, and 3x3 sub-grid, you need to create constraints that ensure no two variables in that row, column, or sub-grid have the same value.

2.3.1 Python implementation

The Sudoku class (contained in the *sudoku.py* file) stores a Sudoku puzzle passed as input to the constructor inside a field called **board** of type `list[list[str]]` (with '.' that represent the empty cells) to easily do computation on the puzzle.

The method `constraint_propagation_solver()` aims to solve the Sudoku contained in `Sudoku.board` using Constraint Satisfaction Problem (CSP) techniques seen in the previous sections. Initially, The `init_domains()` function initializes the domain matrix that is stored inside a `list[list[set]]` variable, where each cell's domain initially contains all values from 1 to 9. The domains are then updated based on the values already given in the Sudoku puzzle. The `update_domains(domains, row_pos, col_pos, value)`

function is used for constraint propagation consequently to the assignment of a new value to a variable. It updates the domains of variables (cells) by removing values that are already present in the domains of the same row, column, or 3x3 sub-grid. This ensures that values are consistent with Sudoku rules.

The main loop used to explore the search tree employs the following data structures:

- **domain_matrix**: of type `list[list[set]]` is used to represent the domains of variables.
- **values_tried**: of type `list[list[set]]` is used to keep track of values that have been tried for each variable.
- **visited_positions**: of type `list[]` is used to maintain a stack of visited positions and their corresponding domains, facilitating backtracking.

The algorithm in the main solver loop selects, at each iteration, the variable within the board with the minimum domain using the function `get_min_domain_cell(...)`, which, given the domains of the variables, returns the position of a cell with the minimum domain. In this way, the solver applies the MRV (Minimum Remaining Values) heuristic. Then, it checks if the domain of the selected variable is empty or not (by subtracting to it also the values already assigned at previous iterations stored in **values_tried**). If the domain is empty the algorithm applies backtracking by picking the last element in **visited_positions** and restoring the variable and the modified domains in the previous step. Otherwise, if the domain is not empty, the algorithm assigns the following value to the variable based on the LCV (Least Constraining Value) heuristic, updates the domains, stores the value inside **values_tried**, and stores the position and modified domains inside **visited_positions**. To apply the LCV heuristic, the algorithm uses the `get_values_ordered_by_least_constraining_value(...)` function, which sorts the values in the given domain based on how many other values they rule out in the domains of neighbouring cells.

The loop continues until there are no more variables with multiple values left in their domains, indicating that a solution has been found.

Finally, the solver takes in input the following parameters:

- **MRV**: boolean that enables or disables the MRV heuristic.
- **LCV**: boolean that enables or disables the LCV heuristic.

2.4 Experiments

In this section, we analyze the performance of the proposed algorithm tested using a database containing real Sudoku puzzles with different difficulties.

2.4.1 Metrics

Two main metrics were recorded for each difficulty level:

1. **Mean execution time**: This metric represents the average time, in seconds, required to solve 100 Sudoku puzzles of a specific difficulty level.
2. **Mean number of backtracks**: This metric represents the average number of backtracks made by the solver to solve 100 Sudoku puzzles of the given difficulty level.

2.4.2 Results

The experiment produced the following results for each difficulty level using both MRV and LCV heuristics:

Difficulty	Mean exe. time (seconds)	Mean num. of backtracks
0	0.0044	295.17
1	0.0064	399.82
2	0.0076	521.1
3	0.0103	675.22
4	0.0112	778.17
5	0.0193	1359.52

Table 2.1: Sudoku Difficulty Levels Analysis with MRC and LCV heuristics

The same experiment executed without the two heuristics produced the following results:

Difficulty	Mean exe. time (seconds)	Mean num. of backtracks
0	0.5194	75453.16
1	0.4446	79032.5
2	0.254	45167.24
3	0.431	72302.88
4	0.3857	64551.39
5	0.5201	94170.04

Table 2.2: Sudoku Difficulty Levels Analysis without MRC and LCV heuristics

2.4.3 Conclusion

The experiment results provide valuable insights into the performance of the Sudoku solver across different difficulty levels:

- As expected, the mean execution time and mean number of backtracks increased as the difficulty of the Sudoku puzzles increased. This is indicative of the increased complexity and depth of search required for more challenging puzzles.

- Difficulty levels 0, 1, and 2 demonstrated relatively quick solving times and a moderate number of backtracks, suggesting that the solver can efficiently handle puzzles of these difficulty levels.
- Difficulty levels 3, 4, and 5 required more time and significantly more backtracks to find a solution, which is consistent with the increased complexity of these puzzles.
- The mean execution time values across all difficulty levels remained relatively low, indicating that the solver is efficient in finding solutions even for challenging puzzles.
- The use of heuristics is very important and reduce drastically the execution time and the search space required to solve a Sudoku puzzle.

Chapter 3

Optimization approach

3.1 Defining optimization problems

Optimization problems are a class of problems that aims to find the best solution or a set of solutions from a given set for possible solutions, often subject to certain constraints. The "best" solution is typically defined based on a specific objective or criteria, which can be either maximizing or minimizing a function or value.

An approach to constraint satisfaction problems is to turn them into an optimization problem by defining an **objective function** that is:

- Positive.
- 0 if and only if all the constraints are satisfied by the given solution.

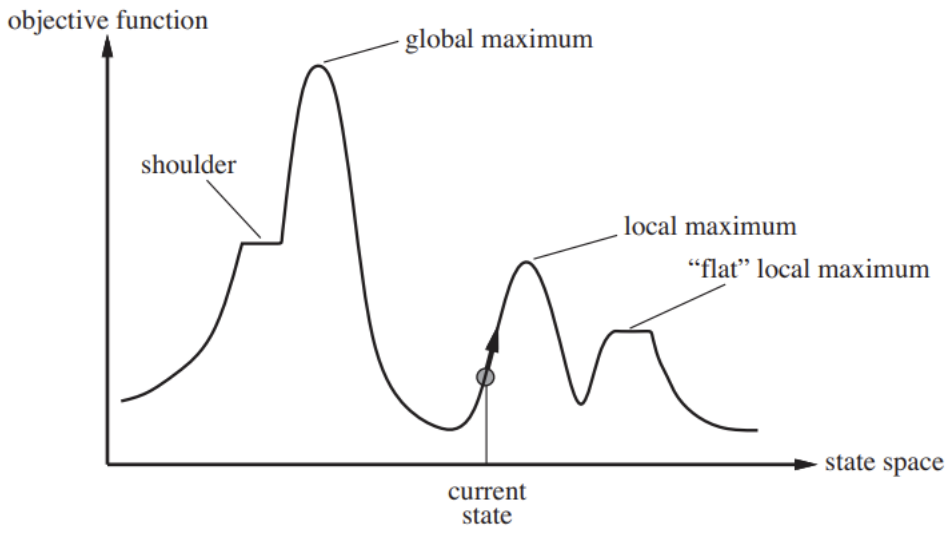


Figure 3.1: A one-dimensional state-space landscape in which elevation corresponds to the objective function.

Figure 3.1 helps us to better the essence of an optimization problem. It illustrates our endeavour to improve the current solution by ascending to the highest point of the objective function, which corresponds to the global maximum. However, it's important to note that optimization landscapes can often be quite complex. They may contain multiple peaks and valleys, including local optima, which are sub-optimal solutions that appear better within their local neighbourhood but are not the best achievable solution

globally. Navigating through these local optima and eventually arriving at the global maximum can be a challenging aspect of optimization problem-solving.

In the following sections, we will see some methods used in the field of optimization to find solutions for complex problems.

3.1.1 Local Search

Local Search is a heuristic method used to solve optimization problems. Unlike complete search algorithms that explore the entire solution space, local search focuses on a single initial solution and seeks to improve it through local iterations.

The basic approach of Local Search is simple: it starts from an initial solution (which can be generated randomly or through other heuristics) and explores part or all of the neighbourhood of that solution, which is the set of solutions that can be obtained through small changes or mutations from the current solution.

Two approaches to local search are possible:

1. In each iteration, explore the entire neighbourhood of a solution and select the best neighbour among all.
2. Alternatively, in each iteration, explore the neighbourhood, updating the current solution with a better neighbour as soon as one is found.

The first case requires more resources and time for the algorithm's execution, while the second is lighter but may not always find the optimal solution in the neighbourhood of a solution.

During each iteration, Local Search examines part of the neighbourhood of the current solution. When a better neighbour of the current solution is found, the current solution is updated with the best neighbour, and the process continues. This process is repeated until a specific termination criterion is met, such as reaching a local minimum, exceeding a certain number of iterations k or finding an optimal solution.

3.1.2 Genetic algorithms

genetic algorithms (or **GA**s) are a class of optimization and search algorithms inspired by the process of natural selection and evolution. They are derived from the Local Search heuristic and are used to find approximate solutions to optimization and search problems by mimicking the principles of biological evolution and genetics.

The process of a genetic algorithm can be summarized as follows:

1. Initialize a **population** of n random candidate solutions to the problem. Each state, or individual, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s.
2. Loop until a specific termination criterion is met, such as reaching a local minimum, exceeding a certain number of iterations k or finding an optimal solution.
   (a) Evaluate each individual's in the population fitness using the **fitness function**. The fitness function is used to evaluate the quality of an individual, by quantifying how close it is to the optimal solution.
   (b) Select n pairs of individuals at random for reproduction, each individual has a weight derived from his fitness value, so individuals with better fitness have a higher probability of being chosen for reproduction. This process is called **selection** and simulates the natural selection mechanism. It favours individuals with higher fitness scores for reproduction.
   (c) For each pair of individuals combine the two individuals (or parents) to create a new individual (or child). This process is called **crossover** and it involves exchanging genetic information between parents to produce new solutions.
   (d) Mutate some of the newly created individuals by introducing random changes to them. This step is called **mutation** and it helps explore the solution space and introduces diversity. Moreover, it ensures that the algorithm does not get stuck in local optima.
   (e) Replace the n old individuals in the population with the n newly generated ones. This step is called **replacement** and aims to create a new population for the next iteration.
3. Return the best solution found in the final population.

Genetic algorithms are known for their ability to explore a large solution space efficiently and are particularly useful for optimization problems where the search space is complex or poorly understood.

3.2 Sudoku puzzle as an optimization problem

As seen before to turn a CSP problem into an optimization problem is mandatory to define an objective function with the following characteristics:

- Positive.
- 0 if and only if all the constraints are satisfied by the given solution.

For the Sudoku puzzle, we define the following objective function:

$$f(s) = \sum_{i=1}^{9} r(i) + \sum_{i=1}^{9} c(i)$$

Where $r(i)$ and $c(i)$ represent the number of missing values in row i or column i in the candidate solution s. Is easy to see that if $f(s) = 0$ then s the columns and rows constraints are all satisfied. To use this objective function we assume that the box constraints are always satisfied for every given candidate solutions.

1	2	3	5	8	4	6	7	9
8	9	7	1	6	3	2	5	4
5	6	4	7	2	9	1	8	3
9	4	6	8	1	5	7	3	2
2	5	8	3	4	7	9	6	1
7	3	1	2	9	6	5	4	8
6	8	5	4	7	1	3	9	5
3	1	9	6	5	8	4	2	7
4	7	2	9	3	2	8	1	6

Figure 3.2: For the following candidate solution $f(s) = 2$.

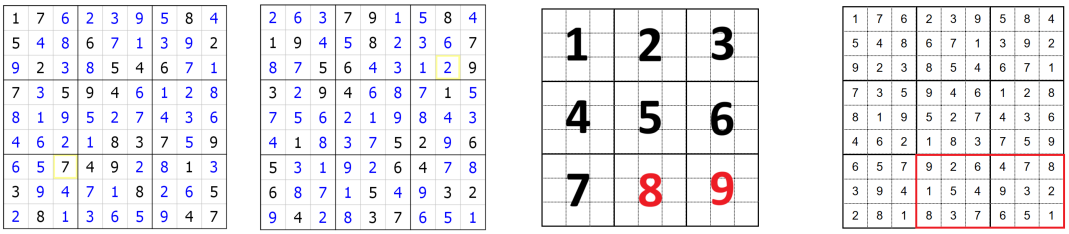
In figure 3.2 the objective function f gives 2 for the candidate solution because we have one missing value in row 7 and another in row 9.

3.2.1 Genetic algorithm solver

In this section, we will focus on the strategies adopted to achieve the steps of GA seen in the previous sections. The strategies are the following:

- **Population**: the population is formed of n individuals and each individual is a candidate solution to the given Sudoku instance. A candidate solution is a complete Sudoku (no empty cells) with the fixed cells of the given instance and other cells filled. An individual can have columns and rows with duplicates whereas all sub-grid must have all the digits from 1 to 9.

- **Fitness function**: The fitness function used is the one shown in the previous section. It sums the number of missing values in each row and column (The sub-grid already satisfies all the constraints for the construction of individuals).
- **Selection**: Each individual in the population has a weight based on its fitness score (less weight implies more probability to be chosen), so n pairs (a, b) with $a \neq b$ are extracted from the population using a sample without replacement of two individuals for n times.
- **Crossover**: To achieve the crossover we must combine parents a, b to create a new valid individual (each sub-grid of the new Sudoku must satisfy the constraint). To do that, we sample the index i of a sub-grid (excluding the first one in the left upper corner) and we create a new Sudoku by attaching the sub-grid from 1 to $i - 1$ of parent a to the sub-grid from i to 9 of parent b. Using this strategy we ensure that all sub-grid have all digit from 1 to 9 and also that the fixed cell of the starting Sudoku instance are maintained.



(a) Parent a (b) Parent b (c) Sampled $i = 8$ (d) New individual

Figure 3.3: Example of crossover for two Sudoku.

- **Mutation**: The adopted strategy chooses at random a sub-grid of the Sudoku and swaps two random cells inside of it. This process is repeated x time with x parameter of the algorithm to test the solver with larger or smaller mutations.
- **Replacement**: the replacement is as described before, all the individuals in the old population are replaced with the new ones.

3.2.2 Python implementation

The Sudoku class (contained in the *sudoku.py* file) stores a Sudoku puzzle passed as input to the constructor inside a field called **board** of type `list[list[str]]` (with '.' that represent the empty cells) to easily do computation on the puzzle.

The method `genetic_solver()` aims to solve the Sudoku contained in `Sudoku.board` using a genetic algorithm. Initially, the function `sudoku_generate_initial_population(...)` is called to randomly generate the initial population, it generates n (with n as function parameter) Sudoku

starting from the one contained in `Sudoku.board` by keeping fixed the initial cells and also by respecting the sub-grid constraint. Then the GA enters a loop and reproduces the steps seen in the previous sections:

1. Using the method `sudoku_fitness(...)` (that takes in input a candidate solution and gives as output its fitness score) compute the fitness of all individuals in the population and save them in a variable of type `list[int]`. Then the GA checks if a valid solution is present in the array (minimum fitness is equal to 0), if it's present exit from the loop and store the solution in `Sudoku.board` otherwise it proceed with the selection step.
2. The GA compute the inverse of the fitness to get the weight of every single individual in the population, then, for a number of times equal to the population size, it randomly chooses two different individuals using `np.random.choice(...)` function.
3. Then, for each pair of parents, the GA calls the function `sudoku_pu_crossover(...)`, which takes in input two individuals and gives as output a new individual following the strategy exposed in the previous section and adds the new individual to the `new_population` variable of type `list[list[int]]`.
4. Every new individual has a small probability p (p is tacked in input to the algorithm) to mutate, so the GA, if `random.random() < mutation_probability` calls the function `sudoku_pu_mutation(...)` passing the given new individual. This function takes in input a Sudoku candidate solution and a number n and applies the mutation strategy described in the previous section for n times to the given Sudoku.
5. Finally, the GA replace the old population with the new one and repeat the loop.

Is important to say that the algorithm does not always succeed in escaping local minima, so after a given number of generations with the same minimum fitness the GA restarts by generating a new random population. The termination criteria of the GA's main loop are the following:

- Maximum number or restarts reached.
- Solution found (fitness equal to 0 in population).

Finally, the GA solver takes in input the following parameters:

1. **population_size**: number of individuals in the population.
2. **mutation_probability**: probability that a new individual has a mutation.

3. **max_restarts**: maximum number of allowed restarts (for loop termination).
4. **num_repetitions_before_restart**: max number of generations with same minimum fitness before restart.
5. **mutation_times**: number of times the mutation strategy is applied for each mutated individual.

3.3 Experiments

In this section, we analyze the performance of the proposed algorithm tested using a database containing real Sudoku puzzles with different difficulties. Due to the randomness component of GAs for this experiment, we tested it on only 10 Sudoku for each difficulty. This is because the algorithm not always finds a solution but sometimes it remains stacked in some local optima.

3.3.1 Metrics

Two main metrics were recorded for each difficulty level:

1. **Mean execution time**: This metric represents the average time, in seconds, required to solve 10 Sudoku puzzles of a specific difficulty level.
2. **Mean number of generations**: This metric represents the average number of generations made by the solver to solve 10 Sudoku puzzles of the given difficulty level.
3. **Number of Sudoku solved**: the number of Sudoku solved for the given difficulty (max 10).

3.3.2 Results

The experiment produced the following results using 2000 as population size, 0.1 as mutation probability, 5 as max number of restarts and 10 as mutation times:

Difficulty	Mean exe. time (s)	Mean num. of gen.	Num. solved
0	54.2321	123.21	10
1	231.253	342.122	7
2	654.2343	1923.452	8
3	443.8753	785.1265	5
4	1234.3857	2043.4452	6
5	2402.123	3212.233	3

Table 3.1: Sudoku Difficulty Levels Analysis using GA solver

3.3.3 Conclusion

The experiment results provide valuable insights into the performance of the Sudoku solver across different difficulty levels:

- As expected, the mean execution time and other metrics increased as the difficulty of the Sudoku puzzles increased. This is indicative of the increased number of local optima present in the landscape of more challenging puzzles.
- The solver does not always solve the given Sudoku and this happens more frequently for more difficult puzzles.
- The mean execution time values across all difficulty levels are very high and variable, indicating that the solver is not efficient in finding solutions.

Chapter 4

Conclusions

In conclusion, we can infer from the two experiments executed respectively for the CSP approach and optimization approach that the best solver is the one that uses the constraint propagation and backtracking algorithm. This algorithm is much faster than the genetic algorithm and also always resolves the assigned instance, on the other hand, the GA solver requires a lot of time to explore the solutions space and due to its randomness sometimes it cannot resolve the puzzle in a reasonable time. In general, Sudoku is not a problem that necessitates the implementation of local search algorithms for its solution.