CA' FOSCARI UNIVERSITY - VENICE

Department of Enviroment sciences, Informatics and Statistics

[CM0623-2] FOUNDATIONS OF ARTIFICIAL INTELLIGENCE (CM90)

Sudoku Solver using Constraint Propagation & Backtracking and Genetic Algorithm

Student:
Michele Lotto 875922

a.y. 2022-23

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

A Sudoku puzzle is composed of a square 9x9 board divided into 3 rows and 3 columns of smaller 3x3 boxes. The goal is to fill the board with digits from 1 to 9 such that:

- each number appears only once for each row column and 3x3 box;
- each row, column, and 3x3 box should containing all 9 digits.

In Figure 1.1 (left) is presented an example for an unsolved Sudoku and in Figure 1.1 (right) is presented its solution by adding the missing red digits.

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

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

Figure 1.1: An unsolved Sudoku (left) and a solved Sudoku (right). [1]

1.1 Assignment

Write a solver for Sudoku puzzles using a constraint satisfaction approach based on constraint propagation and backtracking, and any one of your choice between the following approaches:

- simulated annealing;
- genetic algorithms;
- continuous optimization using gradient projection.

The solver should take as input a matrix where empty squares are represented by a standard symbol (e.g., ".", "_", or "0"), while known square 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
```

The proposal Sudoku solver will take in input a '.txt' file representing the Sudoku using '0' as an empty Cell without spaces. For example:

```
370500006
000360012
000091750
000154070
003070600
050638000
064980000
590026000
200005064
```

The proposal Sudoku solver uses Python 3.10.6 64 bit along with the package 'tabulate' for printing Sudoku as a table for better debugging. Therefore, it is necessary to install the package with "pip3 install tabulate". In order to run the Solver an example main function is presented in file "main.py".

Chapter 2

Solving Sudoku puzzle using a constraint satisfaction approach

2.1 Constraint Satisfaction Problems (CSPs)

A Constraint Satisfaction Problem (CSP) is defined as a triple $\langle X, D, C \rangle$, where [2] :

- $X = \{X_1, \ldots, X_n\}$ is a set of variables;
- $D = \{D_1, \ldots, D_n\}$ is a set of domains of values, one for each variable;
- $C = \{C_1, \ldots, C_m\}$ is a set of constraints.

Each variable X_i can take the values in the nonempty domain D_i. Every constraint $C_j \in C$ is in turn a pair $\langle t_j, R_j \rangle$, where $t_j \subset X$ is a subset of k variables and R_j is a k-ary relation on the corresponding subset of domains D_j. In other words, a constraint is a set of rules on a subset of variables.

2.1.1 Constraint Propagation & Backtracking (CP & BT)

Solving CSP involves some combination of [3]:

- Constraint Propagation: to eliminate values that cannot be part of the solution.
- Search: to explore valid assignments.

Constraint Propagation [3]

Constraint Propagation eliminates values from domain of variables that can never be part of a consistent solution, i.e. values which assignment to the respective variable violates

a constraint or more. For simple problems, this step is enough to obtain a single value per domain, i.e. a solution for the problem. Unfortunately, there are situations where constraint propagation does not guarantee a solution. In general these situations occur when there are too many values in the domains after the CP step. That is attributable to weak constraints and/or too few constraints, which don't reduce effectively the dimension of the domains. In these situation the search in the state-space for solutions is necessary: the simplest approach is pure backtracking.

Backtracking [3]

The state-space can be modelled as a tree. The root is the initial configuration of the problem. A node in this tree is a partial assignment in which some variable are set to a (tentative) value. Pure Backtracking consist in a depth-first search in state-space tree for a solution.

Backtracking is combined with Constraint Propagation: for each tentative assignment in the BT tree, CP eliminates states inconsistent with current hypothesis by local propagating from domains with unique assignments. This approach is called Forward checking. In other words, it previously excludes BT tree branches that will not lead to a consistent solution.

Traditional backtracking uses fixed ordering of variables & values. To optimize the search it's advisable to use one of the following ordering:

- Most Constrained Variable: Pick variable with fewest legal values to assign next.
- Least Constrained Variable: Pick value that rules out the fewest values from neighboring domains.

2.2 Sudoku as a CSP

The Sudoku game can be modelled as a Constraint Satisfaction Problem by defining the $\langle X, D, C \rangle$ triple in the following way:

- $X = \{X_1, \ldots, X_{81}\}$, where X_i is the i-th cell;
- $D = \{D_1, \ldots, D_{81}\}$, where D_i is the domain of values for the X_i cell. Therefore, each D_i is the set of number from 1 to 9.
- C can be defined in two ways:

  a) $C = \{C_1, \ldots, C_{27}\}$ (Direct Constraints), where:
  - $\{C_1, \ldots, C_9\}$ are "Row Constraints": each value in the same row must be different from the others.
  - $\{C_{10}, \ldots, C_{18}\}$ are "Column Constraints": each value in the same column must be different from the others.

- $\{C_{19}, \ldots, C_{27}\}$ are "Square Constraints": each value in the same Sudoku Square must be different from the others.

b) $C = \{C_1, \ldots, C_9\}$ (Indirect Constraints), where C_i: the i-th number must be in a single row, column and Sudoku square only once.

2.2.1 Sudoku as CSP in Python

Python OOP is useful to cast the Sudoku game as an CSP problem. In particular, it is required a way to distinguish between a Cell with a fixed value or an empty Cell and it is also required a way to associate every Sudoku Cell to its value and its domain. For these reasons a Cell class was created containing:

- i: (int): row coordinate in Sudoku board for Cell object. (Useful for debugging).
- i: (int): col coordinate in Sudoku board for Cell object. (Useful for debugging).
- value: (int): value for this Cell. It can take integer values between 1 and 9.
- isEmpty: (bool): identifies if the Cell object represent an empty Cell or not. In other words if the Cell value is fixed or if it can be modified.

If the Cell object represents an empty Cell it also contains:

- domain: (Set[int]): identifies the Cell value domain; it is initialized to the set of integer numbers from 1 to 9.
- visitedDomain: (Set[int]): identifies the values of the Cell domain that have been already visited by Backtracking; it is initialized to the empty set.

The Cell class also contains utility methods that will be used by the CP & BT algorithm:

- getDomainLen(self)->int: it returns the domain length if the Cell is empty, 10 otherwise. This is used to retrieve the minimum domain Cell by CP & BT algorithm. For more details see sub subsection 2.2.2.
- addDomain(self,_n:int)->bool: it adds the specified value to the Cell domain.
- removeDomain(self,_n:int)->bool: it removes the specified value from the Cell domain.
- printDomain(self)->None: it prints the cell domain to stout. (Useful for debugging).
- getCoordinates(self)->tuple[int,int]: it returns the Cell coordinates as a tuple. (Useful for debugging).

Both addDomain(self,_n) and removeDomain(self,_n) check if the Cell is empty before doing operations on its domain, returning 'True' with a successful operation or 'False' otherwise.

When it comes to constraints there is no need to associate them with every Cell. For this reason they were simple stored in the CP & BT algorithm itself.

A Sudoku class was also created to simplify the manipulation of a Sudoku. It contains:

- board: (list [list [Cell]]) : this represent the Sudoku board itself.
- checkSudoku(self)−>bool : utility function that checks if the Sudoku object is correct.
- sudokuSolverCP(self)−>None: (self(Sudoku)->None) : function that solves the Sudoku puzzle with a CP & BT approach.

It also contains some utility function (hidden from the user) used in sudokuSolverCP(self); in particular:

- __minDomain(self)−>Cell: function that return the Sudoku Cell with the minimum length domain.
- __removeDomainAll(self,r:int,c:int,value: int)−>set[Cell]: function that removes the specific 'value' from row of index 'r', columns of index 'c' and square of indexes 'r' and 'c'. It returns a set of Cells from which the 'value' was removed. The return is useful to keep track of modified domains.
- __CP(self)−>None: simple CP in Sudoku. For every full Cell, it removes the value from all the domains of the empty cells present in row, col and square.

2.2.2 Solving Sudoku using CP & BT in Python

The CP & BT solution starts with the initialization of a " visited_cells =LifoQueue()": for each tentative assignment in the BT tree, this queue will keep track of the modified Cell along with a list of Cells which domain was modified by this tentative assignment. This will come in handy in the backtracking step: if the tentative assignment will not get to a solution, it is needed to return to the previous board state by adding the assigned value back to the cells domains from which it was removed and restart by assigning the next value of the domain to the previously modified Cell.

The next step is pure CP by calling the function "__CP(self)−>None". If the Sudoku puzzle is very simple this step is enough to obtain a domain of length=1 for each empty Cell; in this case there will be no backtracking step, but simply an iteration which will assign for each empty Cell the only value in its domain. Unfortunately for the majority of Sudoku puzzles this is not the case.

Backtracking

Backtracking is optimized by using "Most Constrained Variable" ordering, so the next step is retrieving the minimum length domain Cell from the Sudoku board by calling "min_cell =self.__minDomain()". This approach is optimal to store the smallest amount of elements in the queue i.e. minimize the branching factor of the BT tree.

The backtracking step starts with a while loop: "while min_cell.isEmpty" that checks if the Cell which domain has minimum length is a full Cell; if it is the case, then the computation is over: it means that there are no more empty Cells.

Inside the while loop it is checked if "min_cell" has at least one value in its domain that was not previously explored by backtracking; if it is the case then a forward checking step is done, otherwise a nullification of previous forward checking step is done.

Forward checking step: if "min_cell" has at least one value in its domain that was not previously explored by backtracking:

- a value from its domain is assigned to "min_cell" and the assigned value is marked as visited;
- "min_cell" is marked has a full Cell;
- the assigned value is removed from all the domains of the Cells in row, col and square of "min_cell";
- "min_cell" along with a list of Cells which domain was modified by this assignment is en-queue in the "visited_cells" queue;
- finally the next "min_cell" is retrieved for the next iteration.

Nullification of previous forward checking step: If "min_cell" has no values in its domain that was not previously explored then:

- mark each value of "min_cell" domain as not visited;
- retrieve from the queue the last visited Cell ("last_visited_cell") and the Cells in which the domain was modified by its assignment ("domainRemovedCells");
- mark the last visited Cell as empty;
- for each Cell in "domainRemovedCells", add the last visited Cell value to their domain;
- set "min_cell=last_visited_cell" for the next iteration.

The full solution code is reported in Listing 2.1

```
1  def sudokuSolverCP(self):
2
3    #Queue of visited cells
4    visited_cells=LifoQueue()
5
6    #initial constraint propagation
7    self.__CP()
8
9    #retrieve the min cell domain
10   min_cell=self.__minDomain()
11
12   #if the min domain cell is a full cell then the computation is over
13   while min_cell.isEmpty:
```

```
14
15      #if there is at least one value that was not previously assigned
16      #in the min domain cell domain then:
17      if len(min_cell.domain-min_cell.visitedDomain)>0:
18
19        #1) update value of the min domain cell
20        min_cell.value=(min_cell.domain-min_cell.visitedDomain).pop()
21        min_cell.visitedDomain.add(min_cell.value)
22
23        #2) update min domain cell to a full cell
24        min_cell.isEmpty=False
25
26        #3) update domains of row, col and square by removing
27        #the value assigned to the min domain cell
28        domainRemovedCells=self.__removeDomainAll(min_cell.i,
29                                                  min_cell.j,
30                                                  min_cell.value)
31
32        #4) add the min cell and the cells in which the domain
33        #is modified to the visited cells queue as a tuple
34        visited_cells.put((min_cell,domainRemovedCells))
35
36        #5) retrieve the next min domain cell
37        min_cell=self.__minDomain()
38
39      #if the cell domain is empty or if all the values
40      #in the cell domain were previously assigned backtracking:
41      else:
42
43        #1) reset visited domain for min domain cell
44        min_cell.visitedDomain=set()
45
46        #2) get last visited cell and the cell in which
47        #the domain was modified by its assignment
48        last_visited_cell,domainRemovedCells=visited_cells.get()
49
50        #3) update last visited cell to a empty cell
51        last_visited_cell.isEmpty=True
52
53        #4) add the value previously assigned to the last visited cell
54        #to the cells previously modified by its assignment
55        for cell in domainRemovedCells:
56          cell.addDomain(last_visited_cell.value)
57
58        #5) set the min domain cell as the last visited cell to explore
59        #the next value of its domain
60        min_cell=last_visited_cell
```

Listing 2.1: sudokuSolverCP

Chapter 3

Solving Sudoku puzzle using an optimization approach

An optimization problem is the problem of finding the optimal solution from all feasible solutions of a given problem [4]. Solutions are compared by using an objective function, that gives a score to all the feasible solutions. The optimal solution has the maximum or minimum score. A possible approach for Constraints Satisfaction problems is to cast them into optimization problems [5].

3.1 Local search

Local search is a heuristic method for solving computationally hard optimization problems. Local search can be used on problems that can be formulated as finding a solution maximizing or minimizing the objective function among a number of candidate solutions [6] [5].

Local search algorithms move from a solution to an adjacent solution in the space of candidate solutions (the search space) by applying local changes (i.e. by maximizing or minimizing the objective function), until an optimal solution is found. This approach requires the notion of an adjacent solution and a strategy to decide on which of the available solutions to move [6] [5].

Local search approach can lead to a local optimal solution that could not be a global solution i.e. the best possible solution, see Figure 3.1 [5]. Indeed a local solution could have a score very close to the best possible score, but the local solution itself could be very different from the best possible solution. For this reason local search algorithms need to have strategies for escaping the local optima and converge to the global optimum [5].

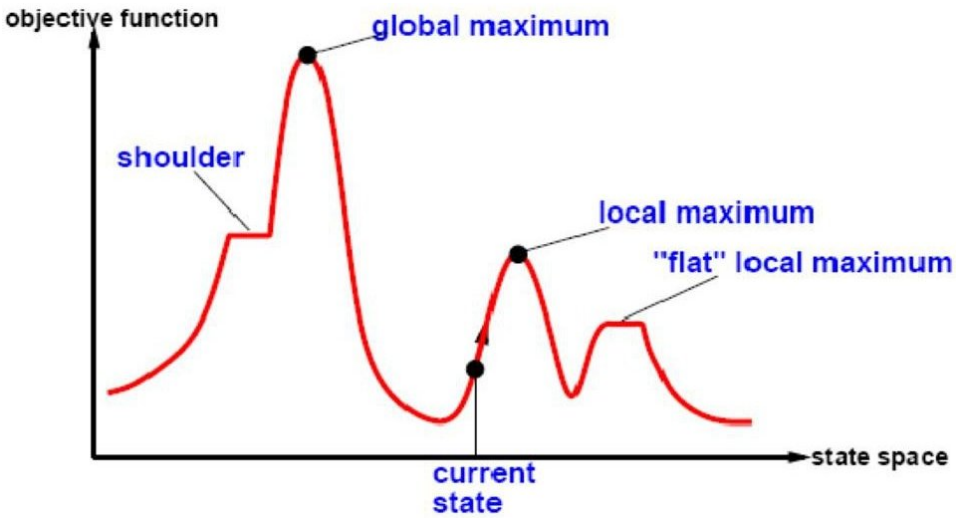

Figure 3.1: Plot of the objective function scores for each solution in the search space (state space). [5]

3.1.1 Genetic Algorithms (GAs)

From the local search heuristic different approaches can be derived, one of them is called "Genetic algorithm", inspired by the process of natural selection [5] [7]. In a GA, a population of candidate solutions (individuals) to an optimization problem is evolved toward better solutions [7]. An individual is represented as a string over a finite alphabet (often binary strings, but other representations are also possible) [7] [5]. The fitness function is the GA equivalent of objective function, usually the higher fitness value of an individual is, the better the individual is [5].

Steps of Genetic Algorithm [7]:

1. Start: Generate random population of n individuals (suitable solutions for the problem).
2. Fitness: Evaluate the fitness function of each individual in the population.
3. Selection: Select a certain percentage of best individuals from the population according to their fitness and a certain percentage of random individuals.
4. Crossover: Until there are n individuals again in the population, with a certain probability, combine two individuals (parents) to form a new individual (child).
5. Mutation: Alterate a certain percentage of individuals with a certain probability. This step consents to escape the local optima by introducing some randomness in some individuals.
6. Test: If the end condition is satisfied, stop and return the best solution in current population.

7. Loop: Go to step 2.

Using a GA requires choosing the right strategy of implementation of these fundamental steps [5] [7]:

- Fitness: The fitness function ranks the individuals by a specific propriety they have, which will be maximized by the GA. Choosing the right propriety is fundamental to avoid getting stuck on local maximum: two individual with the same fitness could be extremely different from each other.
- Selection: Darwinian selection holds that "Stronger" individuals have greater probability of surviving and of reproducing. In the context of GA, stronger individuals are those with higher fitness. The simplest selection approach is called "Roulette wheel selection": let f_i be the fitness values of i, the probability that i is selected for reproduction is:
$$p_i = \frac{f_i}{\sum_k f_k}$$
Other approaches are also common and their choice depends on the problem itself.
- Crossover: In this step choosing a fair crossover point is fundamental for the convergence of the algorithm: a wrong crossover point could annihilate the progress done so far or slow down the progression to the optimal solution. Its choice depends on the representation of individuals and also on the problem itself (a wrong crossover point could violate a problem constraint previously satisfied by the parents). In general, it is important to combine at least some information from each of the two parents, otherwise a simple copy of one of the two parents is done, other general considerations are not possible and depend on the specific problem.
- Mutation: Choosing the mutation strategy is fundamental to escape the local optima. How to mutate an individual depends on how it is represented and on the problem itself. Common choices are swapping some parts of the individual representation or randomize some parts of the individual. It also requires a fine tuning of the mutation probability.

3.2 Solving Sudoku puzzle using a GA

In section 3.1.1, it is discussed the importance of the choice of the right strategy for the fundamental steps of GAs, therefore it is mandatory to present the choices taken before venturing in the algorithm itself:

- Individual: In the Sudoku game context an individual is defined as a Sudoku which full Cells are equal to the full Cells of the start Sudoku; empty cells differs. Each row of an individual must have all the 9 digits in it; columns and Sudoku squares can contain duplicates.

- Fitness: The fitness evaluation chosen for Sudoku counts the number of satisfied constraints for each column and Sudoku square (the rows already satisfy the constraints thanks to the individual definition). Each column and Sudoku square has 9 constraints: one for each possible digit. Therefore the number of satisfied constraints is calculated as:

$$9 - \#(D - A)$$

where D is defined as $D = \{1,2,...,9\}$ (the set of all the possible digits) and A as the set of digits present in the column or Sudoku square of interest. The number of satisfied constraints for each of these objects is summed up to obtain the total number of satisfied constraints for the given Sudoku board. A Sudoku with a fitness of 162 out of 162 satisfies all the constraints (9x9+9x9). An example is reported in Figure 3.2: the yellow and red Cells are randoms digits. The red ones are wrong. The first column satisfies 7 out of 9 constraints: 4 and 8 has duplicates (digits 2 and 3 are not present).

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

Figure 3.2: Sudoku with fitness of 114 out of 162.

- Selection: The classical roulette wheel selection approach did not produce the desired results: too few best individual were selected, resulting in a too slow convergence of the algorithm. For this reason a custom approach was chosen: a certain percentage of population is chosen randomly and a certain percentage of the population is chosen starting by the best individual among all. This approach guarantees that the k best individuals are always chosen, and also guarantees some randomness in the selection.

- Crossover: The crossover point needs to guarantee that the generated Sudoku is a valid individual i.e. each row of an individual must have all the 9 digits in it. For this reason the choice of the crossover point was random number from 1 to 8, representing the number of consecutive rows from parent 1 to be part of the new Sudoku (child);

the missing rows of the child are taken from parent 2. This crossover point also guarantees that at least one row is taken from parent 1 and parent 2. For example suppose that the crossover point is 6, then the first 6 rows of parent 1 will be be first 6 rows of the child and the remaining 3 rows are taken by parent 2. This example is reported in Figures 3.3 and 3.4

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

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

Figure 3.3: Parents Sudokus

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

Figure 3.4: Child Sudoku

- Mutation: The mutation strategy is swapping a fixed random number of pairs of Cells in a fixed random number of row of the given Sudoku. An example is reported in Figure 3.5: on the left picture are randomly selected 2 pairs of Cells of 2 random selected rows; on the right these pairs are swapped.

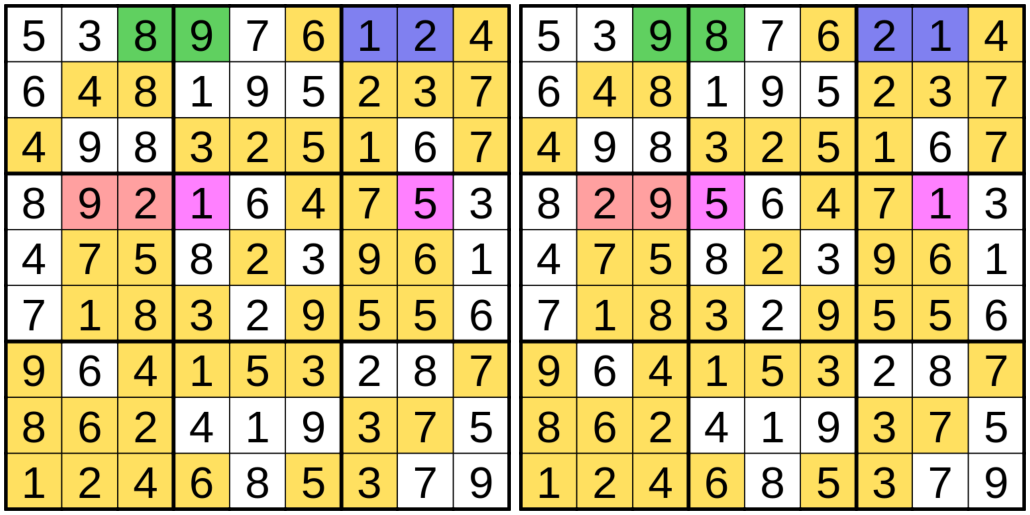

Figure 3.5: Sudoku mutation

- Restart: The GA could get stuck in a local maximum and could never reach the global maximum, even with a decent mutation step. For this reason it has been chosen that after a number of generation without improvement of the best individual, the computation is restarted.

3.2.1 Python implementation

The OOP structure explained in subsection 2.2.1 has been expanded in order to use GAs. In particular the Sudoku object have additional parameters and methods:

- satisfied_constraint : (int) : this parameter represents the fitness score for the given Sudoku object.

- __fitness(self)->None: function that calculates or updates the fitness score for the given Sudoku object.

- __randomizeSudokuAndScore(self): function that randomize the empty Cells of the given Sudoku object. At the end of its execution it calls '__fitness(self)->None' to update the fitness score of the modified Sudoku object.

- __getChild(parent1:Sudoku,parent2:Sudoku)->Sudoku: static function that given 2 Sudoku objects (parents) returns a new Sudoku (child), which is a combination of the

two (crossover function). At the end of its execution it calls '__fitness(self)->None' to set the fitness score of the new Sudoku object.

- __mutation(self, n_rows, n_cells_per_row)->None: function that mutates the given Sudoku object: swap ncells_per_row randomly in nrows random rows. At the end of its execution it calls '__fitness(self)->None' to update the fitness score of the modified Sudoku object.
- __isSolution(population:list[Sudoku])->Sudoku: static function that checks if in the given population a Sudoku is a valid solution i.e. if a Sudoku has the maximum score.
- sudokuSolverGA(self, ...)->None: function that solves the given Sudoku object using a GA approach. The others parameters of this function are:
  - population_size:int=3000: the size of the initial random generation.
  - selection_rate: float=0.25: the percentage of best individuals selected.
  - random_selection_rate:float=0.25: the percentage of individuals selected randomly.
  - n_children:int=4: the number of children generated by the same two parents Sudokus.
  - mutation_rate:float=0.3: the percentage of individuals that are going to be mutated.
  - n_rows_swap:int=3: the number of row in which n_cells_per_row_swap pairs of Cells will be swapped.
  - n_cells_per_row_swap:int=1: the number of pairs of Cells that will be swapped within the same row.
  - n_generations_no_improvement:int=30: the number of generation without improvements admitted before restarting the computation.

  These parameters were fine tuned to obtain the maximum performance.

A schematic representation of sudokuSolverGA(self, ...)−>None function in reported in Listing 3.1.

```
def sudokuSolverGA(self, ... ):

  while True:

    #0) Initial generation phase

    while True:

      #1) Selection phase

      #2) Crossover phase

      #3) Mutation phase

      #4) Evaluation phase

      #5) (optional) Restart phase
```

Listing 3.1: sudokuSolverGA

Initial generation phase

In the 'Initial generation phase', 'population_size' individuals are generated by the initial Sudoku as shown in Listing 3.2. The presence of a solution is also checked, despite it is very unlucky to happen.

```
old_population=[Sudoku(self) for x in range(population_size)]
for sudoku in old_population:
    sudoku.__randomizeSudokuAndScore()

solution=Sudoku.__isSolution(old_population)
if solution is not None:
    print("solution found at initial generation (generation 0)")
    self.board=copy.deepcopy(solution)
    return
```

Listing 3.2: Initial generation.

Selection phase

In the 'Selection phase', 'population_size∗random_selection_rate' individuals are chosen randomly by using the standard 'sample' Python function and 'population_size∗selection_rate' best individuals are selected by sorting the population list by 'satisfied_constraint' (Sudoku object attribute). The selected population is stored in 'population' variable. The code is reported in Listing 3.3.

```
#random selection
population=sample(old_population,
                  int(population_size*random_selection_rate))

#selection
old_population.sort(key=operator.attrgetter("satisfied_constraint"),
                  reverse=True)

for x in range(int(population_size*selection_rate)):
    population.append(old_population[x])
```

Listing 3.3: Selection.

Crossover phase

The 'Crossover phase' starts with a copy of the selected population in a new list, these individuals will be in the next generation. While the new population size doesn't reach the target population size, 2 individuals are sampled from population for generating n_children new individual (children) by the crossover function. At the end of the while loop the new population will contain the selected individuals and the individual generated by the selected individuals. The code is reported in Listing 3.4.

```
new_population=[copy.deepcopy(s) for s in population]

while(len(new_population)<population_size):

    children=[]

    parent1,parent2=sample(population,k=2)

    for _ in range(n_children):
        child=Sudoku.__getChild(parent1,parent2)
        children.append(child)

    new_population+=children
```

Listing 3.4: Crossover.

Mutation phase

In the 'Mutation phase', the mutation function is applied to population_size∗mutation_rate individuals of the new population. The shuffle(new_population) function call guarantees that the mutated individuals are chosen randomly. The code is reported in Listing 3.5.

```
shuffle(new_population)

for e in range(int(population_size*mutation_rate)):
    new_population[e].__mutation(n_rows_swap,n_cells_per_row_swap)
```

Listing 3.5: Mutation.

Evaluation phase

In the 'Evaluation phase', some statistics are generated to evaluate the current generation of Sudokus. After the 'Initial generation phase' two variables are initialized as shown in Listing 3.6:

- restart: a counter for the number of generation without improvement.
- best_fit: the number of satisfied constraint for the best individual in the population.

```
restart=0
best_fit=max(old_population,
             key=operator.attrgetter("satisfied_constraint")
             ).satisfied_constraint
```

Listing 3.6: Initialization of evaluation variables.

After the 'mutation phase', it is checked that there is a valid solution among the individuals of the current population. If it is the case then the computation is over and the Sudoku object board is set as the solution one. If this is not the case, it is checked than the current best individual is an improvement respect to the previous best individual. If it is the case then restart is set to '0' and the new best individual is set to be the current one. At the end of this phase restart is increased by one and old_population is set as new_population for the new iteration. The code is reported in Listing 3.7.

```
fit=max(new_population,
        key=operator.attrgetter("satisfied_constraint")
        ).satisfied_constraint

solution=Sudoku.__isSolution(new_population)
if solution is not None:
    self.board=copy.deepcopy(solution.board)
    return

if fit>best_fit:
    best_fit=fit
    restart=0

old_population=new_population
restart+=1
```

Listing 3.7: Evaluation.

Restart phase

In the 'Restart phase', it is checked if restart is greater than the number of generations without improvements. If this is the case, then the computation is restarted from the initial generation. The old variables are cleaned for better performance. The code is reported in Listing 3.8

```
if restart>n_generations_no_improvement:

    del population,new_population,old_population,
    children,child,parent1,parent2
    gc.collect()

    break
```

Listing 3.8: Restart.

Chapter 4

Benchmark and Results

The two approaches described in the previous chapters have been tested using a data-set of 22 Sudokus of different difficulty: easy (35-41 full cells), normal (29-32 full cells), medium (25-29 full cells) and hard (21-25 full cells).

The test where done using a Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processor along with Python 3.10.6 64bit.

4.1 Constraint Propagation & Backtracking

The CP & BT approach is the one that had the best results. In Tables 4.1, 4.2, 4.3 and 4.4 for each Sudoku in the data-set is reported the number of full cells, the execution time and the number of restored nodes of the BT tree. As expected, easy Sudokus have been solved only with the Constraint Propagation step or with very few restored nodes in the BT tree. More difficult Sudoku requires the Backtracking step.

name	full cells	execution time	restored nodes
easy1	35	0.0015s	0
easy2	37	0.0015s	0
easy3	38	0.0016s	0
easy4	40	0.0015s	0
easy5	41	0.0015s	0
easy6	35	0.0017s	0

Table 4.1: CP & BT results for easy Sudokus.

name	full cells	execution time	restored nodes
normal1	29	0.0021s	11
normal2	30	0.0016s	0
normal3	32	0.0018s	3
normal4	32	0.0017s	3
normal5	30	0.013s	316

Table 4.2: CP & BT results for normal Sudokus.

name	full cells	execution time	restored nodes
medium1	29	0.0035s	75
medium2	25	0.0033s	5
medium3	25	0.0018s	2
medium4	28	0.0032s	56
medium5	25	0.0017s	0

Table 4.3: CP & BT results for medium Sudokus.

name	full cells	execution time	restored nodes
hard1	23	0.0846s	3204
hard2	25	0.0024s	29
hard3	23	0.0017s	2
hard4	22	0.0022s	0
hard5	21	0.0018s	0
hard6	21	0.0594s	1313

Table 4.4: CP & BT results for hard Sudokus.

In Table 4.5 are reported mean and standard deviation for the execution time and number of restored nodes in the BT tree.

name	mean	standard deviation
execution time	0.0089 s	0.02094 s
restored nodes	228.1364	722.2385

Table 4.5: Mean and standard deviation for the execution time and number of restored nodes.

In Figure 4.1 is reported the correlation matrix for the number of full cells, the execution time and the number of restored nodes. The correlations between full cells and the other

two variables are weak and negative. The number of full cells is not significant to explain the variability of the execution time or the number of restored nodes: this means that ranking Sudoku difficulty by the number of full cells is not a good metric for this type of algorithms. The correlation between execution time and restored cells is very strong and positive. Given the algorithm structure, this result is not surprising: for each restored node the algorithm needs to go one step back and try another path; obviously this process increases the execution time.

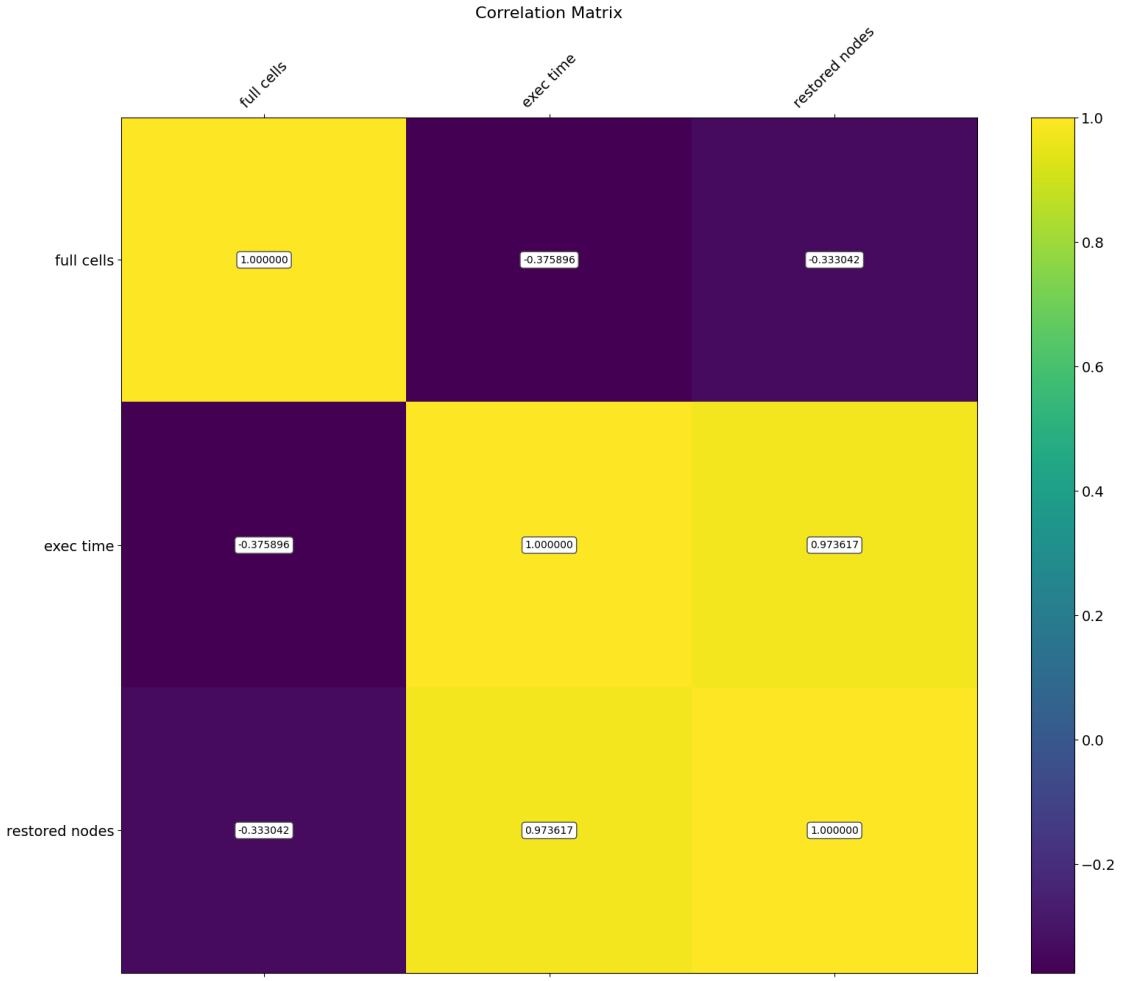

Figure 4.1: Correlation matrix for CP & BT

4.2 Genetic Algorithm

The Genetic Algorithm approach is the one that had the worst results. In Table 4.6, 4.7, 4.8 and 4.9 for each Sudoku in the data-set is reported the execution time, the number of full cells, the number of restarts and the number of total generation (for each restart). Given

the strong randomness component of GAs it was chosen to repeat the test 3 times and report only the median test results. As the Sudoku difficulty increases a mean increment in the execution time along with the number of restarts and the number of total generations is registered. This is not always the case because these 3 variables depends on the number of local maximum that the given Sudoku has; in other words the probability of a restart increases (the algorithm is more likable to get stuck) as the number of local maximum increases.

name	full cells	execution time	restarts	total generations
easy1	35	47.1244 s	1	47
easy2	37	36.1862 s	1	25
easy3	38	40.6885 s	1	29
easy4	40	41.8533 s	1	30
easy5	41	29.6387 s	1	20
easy6	35	54.0527 s	1	40

Table 4.6: GA results for easy Sudokus.

name	full cells	execution time	restarts	total generations
normal1	29	198.5801 s	2	153
normal2	30	477.2946 s	5	370
normal3	32	573.0072 s	6	448
normal4	32	147.4415 s	2	112
normal5	30	666.5298 s	7	514

Table 4.7: GA results for normal Sudokus.

name	full cells	execution time	restarts	total generations
medium1	29	196.7747 s	3	131
medium2	25	1212.9705 s	11	938
medium3	25	1405.7452 s	14	1100
medium4	28	134.7545 s	2	106
medium5	25	1068.3213 s	11	857

Table 4.8: GA results for medium Sudokus.

name	full cells	execution time	restarts	total generations
hard1	23	3442.7146 s	23	2030
hard2	25	391.1148 s	3	271
hard3	23	3423.0165 s	20	1738
hard4	22	4187.7347 s	32	2852
hard5	21	4187.7347 s	35	3141
hard6	21	20s	8	8

Table 4.9: GA results for hard Sudokus.

In Table 4.10 are reported mean and standard deviation for the execution time, the number of restarts and the number of total generations.

name	mean	standard deviation
execution time	1336.0945 s	1919.1786 s
restarts	10.4090	13.0700
total generations	926.4091	1368.0832

Table 4.10: Mean and standard deviation for the execution time, the number of restarts and the number of total generations.

In Figure 4.2 is reported the correlation matrix for the number of full cells, the execution time, the number of restarts and the number of total generations. The correlations between full cells and the other three variables are important and negative. In this case, the number of full cells is significant to explain the variability of the execution time and the number of restored nodes: this means that ranking Sudoku difficulty by the number of full cells is a good metric for this type of algorithms. The correlation between the number of restarts and the number of total generations is very strong and positive: it is easy to understand that restarting the algorithm means having more total generations. The execution time is strongly and positively correlated with both the number of restarts and the number of total generations. Given the algorithm structure, this result is not surprising: each generation corresponds to an iteration of the internal while loop (selection, crossover, mutation and evaluation phases) each of which is very time demanding.

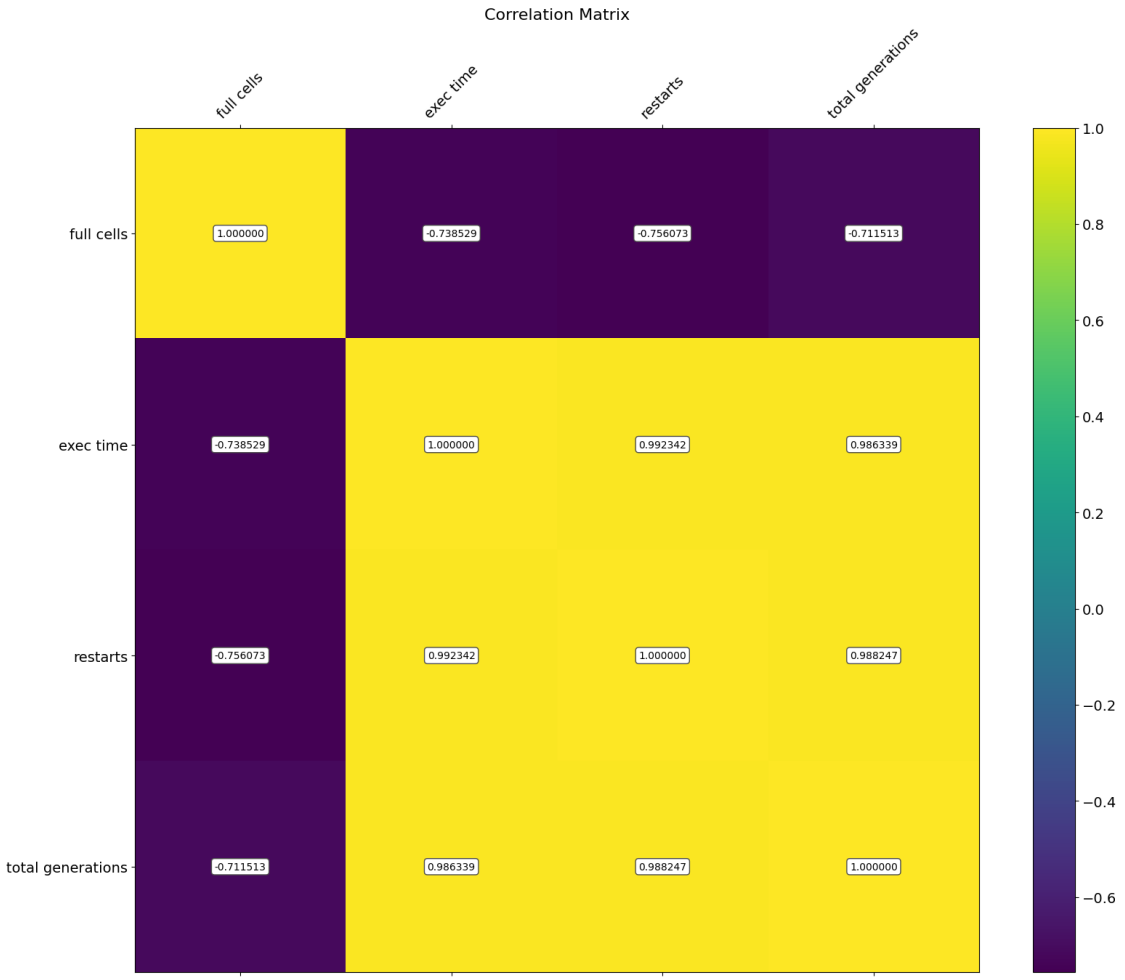

Figure 4.2: Correlation matrix for GA

Given that the Sudoku difficulty is a good measure for ranking Sudokus other analysis are possible. In figure 4.3 is reported the box plot by difficulty and execution time: increasing the difficulty means increasing both the execution time mean and variability. In figure 4.4 is reported the box plot by difficulty and total generations: increasing the difficulty means increasing both the execution time mean and variability.

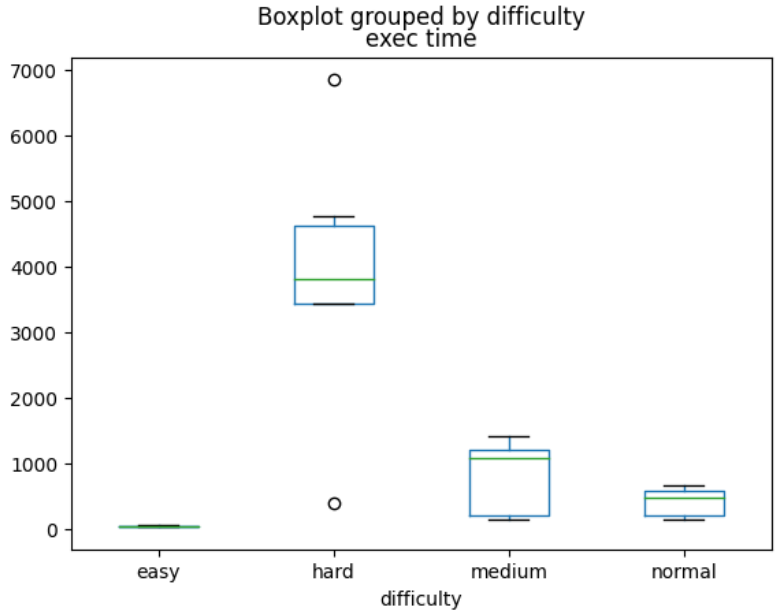

Figure 4.3: Box plot by difficulty and execution time.

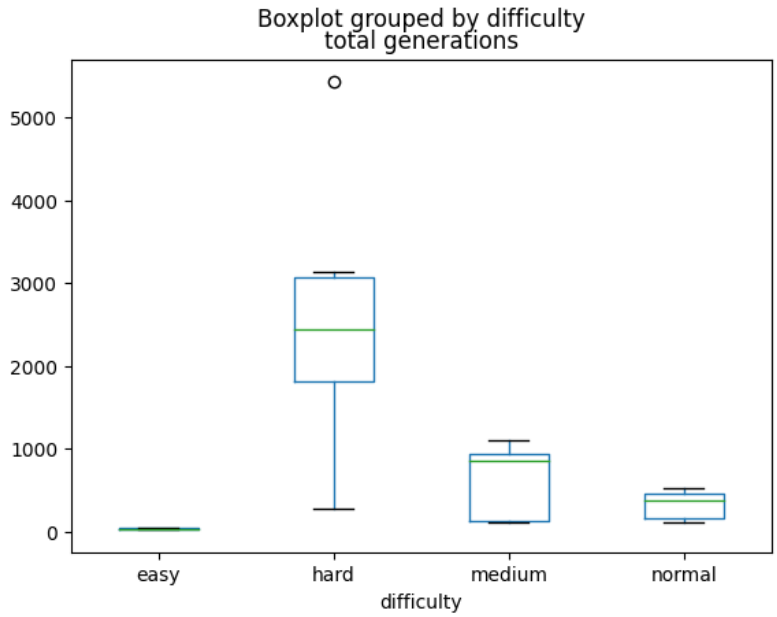

Figure 4.4: Box plot by difficulty and total generations.

Chapter 5

Conclusions

In this work two approaches for solving a Sudoku were implemented and discussed. The Constraint Propagation and Backtracking algorithm has given the best results, solving a Sudoku problem with a mean of 0.0089 seconds and a standard deviation of 0.02094 seconds. This large variability is due to the variability of restored nodes since these two variables are correlated. The Genetic Algorithm has given the worst results, solving a Sudoku with a mean of 1336.0945 seconds and a standard deviation of 1919.1786 seconds. Compared with CP & BT approach, this one is very slow. Surprisingly with this approach the number of full cells is a good ranking metric for this algorithm. Given these considerations the Constraint Propagation & Backtracking approach is the best choice between the two.

Bibliography

[1] *Sudoku*. Wikipedia. URL: https://it.wikipedia.org/wiki/Sudoku.

[2] Stuart Jonathan Russell and Peter Norvig. "Artificial Intelligence: A Modern Approach". In: Prentice Hall, 2010. Chap. 6.

[3] Andrea Torsello - Ca' Foscari University. "Constraint Satisfaction slides".

[4] *Optimization Problem*. Wikipedia. URL: https://en.wikipedia.org/wiki/Optimization_problem.

[5] Andrea Torsello - Ca' Foscari University. "Local Search slides".

[6] *Local search (optimization)*. Wikipedia. URL: https://en.wikipedia.org/wiki/Local_search_(optimization).

[7] *Genetic Algorithm*. Wikipedia. URL: https://en.wikipedia.org/wiki/Genetic_algorithm.