Ca' Foscari
University
of Venice

Foundation of Artificial Intelligence

ASSIGNMENT 1

Professor
Andrea Torsello

Student
Pierluigi Marchioro
Matriculation Number 881929

Academic Year
2022-23

Contents

0 Introduction

This report describes the theory and the software implementation of the techniques used to achieve the goal of the assignment, better highlighted below, as well as giving a brief introduction on what the Sudoku game consists in.

Assignment Goal 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

Chosen Approaches The approaches discussed in this report will be the mandatory *Constraint Propagation and Backtracking* as well as *Simulated Annealing*. Their software implementation was made in the Python programming language, and some code snippets about the core parts of the solver implementations will be shown throughout the report.

1 The game of Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle, based on a 9 x 9 grid. The rules are pretty simple: each cell in the grid must be filled with a digit from 1 to 9 so that each column, row and 3 x 3 subgrid (also called "square" or "box") contains exactly one occurrence of each digit.

The game starts from a partially completed grid, such as the one below, and the player must try to complete it by respecting the aforementioned rules.

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

Figure 1.1: Starting, partially completed, Sudoku grid

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 1.2: Solution to the above presented Sudoku

2 Constraint Propagation

2.1 Theoretical Foundations

2.1.1 Introduction to Constraint Programming

Constraint programming is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints which must be satisfied by the solution.

But, then, what is a constraint? It is simply a logical relation among several unknowns (or variables), each taking a value in a given domain. A constraint thus restricts the possible values that variables can take; it represents some partial information about the variables of interest. Naturally, constraint programming problems have more than one constraint, and, to complicate things further, constraints are rarely independent.

2.1.2 Constraint Satisfaction

Currently there can be seen two branches of Constraint Programming research which arise from distinct bases and, thus, use different approaches to solve constraints. Constraint Satisfaction is one of the two branches, the other being Constraint Solving. For the purposes of this report, the focus will be on the former.

Constraint Programming divides in two branches, namely *Constraint Satisfaction* and *Constraint Solving*, which use different approaches to solve constraints. For the purposes of this report, the focus will be on the former, and specifically on *Constraint Satisfaction Problems (CSPs)*

Constraint Satisfaction Problem (CSP) A Constraint Satisfaction Problem (CSP) is defined mainly in three terms:

- **Variables**: entities that can have multiple values assigned to them
- **Domains**: set of values that can be assigned to a variable
- **Constraints**: conditions that define what is considered a valid assignment to a variable; each constraint restricts the combination of values that a set of variables may take simultaneously

The solution of a CSP consists in assigning to each variable a value that satisfies all the constraints. Thus, the CSP is a combinatorial problem which can be solved by search. In the following sub-sections, basic systematic search methods, as well as techniques that improve on the latter, are discussed.

2.1.3 Basic Systematic Search

There are two main classes of systematic search algorithms.

- algorithms that search the space of complete assignments, i.e., the assignments of all variables, till they find the complete assignment that satisfies all the constraints
- algorithms that extend a partial consistent assignment to a complete assignment that satisfies all the constraints.

Backtracking (BT) belongs to the latter class. In the BT method, variables are instantiated sequentially and as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial assignment violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Clearly, whenever a partial assignment violates a constraint, backtracking is able to eliminate a portion of the search space, because the constraint-violating partial assignment will never be performed again.

2.1.4 Consistency techniques

The late detection of inconsistency is a main shortcoming of the BT paradigm. It is for this reason that various consistency techniques were introduced to overcome this disadvantage and preemptively prune the search space. Such techniques range from simple node-consistency and the very popular arc-consistency to full, but expensive path consistency. For the purposes of this assignment, the most interesting techniques to delve into were the former two, that is, node consistency and arc consistency.

Node Consistency The node representing a variable X in a constraint graph is *node consistent* if and only if for every value V in the current domain D_X of X, each unary (meaning that it concerns only that variable) constraint on X is satisfied.

Arc Consistency An arc (V_i, V_j) is *arc consistent* if and only if, for every value x in the current domain of V_i which satisfies the constraints on V_i there is some value y in the domain of V_j such that $V_i = x$ and $V_j = y$ is permitted by the binary constraint between V_i and V_j.

2.1.5 Constraint Propagation

By integrating systematic search algorithms with consistency techniques, it is possible to get more efficient constraint satisfaction algorithms. Therefore, a third possible schema was introduced that embeds a consistency algorithm inside a search algorithm. Such schemas are usually called look-ahead strategies and they are based on idea of reducing the search space through constraint propagation.

Simple backtracking (BT) already performs some kind of consistency technique, which can be summarized as arc consistency checks between a newly instantiated (or updated) variable and already instantiated ones. As mentioned before, however, the inconsistency is detected late, at the last moment, highlighting the fact that there is a lot of room for search space reduction.

Forward checking is the easiest way to achieve such a feat. The forward checking algorithm is based on the idea that future assignments cannot conflict with current ones. Arc consistency

checks are performed between pairs of a not-yet instantiated variable and an instantiated one, as opposed to the simple BT algorithm that considers only pairs of already instantiated variables. Therefore, the invariance that *for every un-instantiated variable there exists at least one value in its domain which is compatible with the values of instantiated variables* is maintained. Consequently, forward checking allows branches of the search tree that will lead to a failure to be pruned earlier than with backtracking.

2.1.6 Ordering

The efficiency of search algorithms which incrementally attempts to extend a partial solution depends considerably on the order in which variables are considered for instantiations. Usually, a static ordering of variables performs worse than a dynamic one, hence it is useful to consider greedy techniques such as choosing the variable with the smallest or biggest domain (most or less constrained) first.

2.1.7 CSP Theory Applied to Sudoku

As mentioned in the first chapter, Sudoku is a problem with clearly defined constraints, that is, for each variable, which is represented by a Sudoku cell, its value must not be assigned to other cells of the same row, column or box. It is therefore very intuitive to try to solve this problem by applying *Forward Checking*: the domain of each cell, that is, the set of values available for assignment, must not contain values that are already assigned to cells of the same row, column or square. This way, future assignments are guaranteed to not violate any constraint, hence greatly reducing the search space.

2.2 Software Implementation

The core parts of the implementation are the main function, which implements the backtracking part of the solver, the way the forward checking and the domain updates are performed, and the chosen variable ordering technique. All of these aspects, as well as the fundamental data structures, are discussed in detail in the following sub-sections.

2.2.1 Fundamental Data Structures

2.2.1.1 CellCoordinates Class

The *CellCoordinates* class simply represents the pair of row and column indexes of some Sudoku cell. This class is widely used in the solver implementation, possibly paired with an integer value, to represent a full Sudoku cell.

```
pierluigimarchioro
@dataclass(frozen=True)
class CellCoordinates:
    row: int
    col: int

    pierluigimarchioro
    def __str__(self):
        return f"(row: {self.row}, col: {self.col})"
```

Figure 2.1: *CellCoordinates* Class

2.2.1.2 SudokuGrid Class

The *SudokuGrid* class provides a basic interface to access the cells of a Sudoku grid, that is, getters and setters for the whole grid, for single cells, or for whole rows, columns and squares. Additionally, this class serves as super-type for some more specific implementations that provide functionality related to a specific solving technique, like the later discussed *ConstraintPropagationSudokuGrid*.

```
pierluigimarchioro
class SudokuGrid:
    _inner_grid: np.ndarray
    """
    9x9 Sudoku grid
    """

    empty_cell_marker: int
    """
    Value used to mark empty cells in the Sudoku grid
    """

    pierluigimarchioro
    def __init__(self, starting_grid: np.ndarray = None, empty_cell_marker: int = -777):...
```

Figure 2.2: *SudokuGrid* Class (1) - The actual Sudoku grid is represented by a *numpy* two-dimensional array

```
pierluigimarchioro
def get_value(self, cell: CellCoordinates) -> int:...

pierluigimarchioro
def set_value(self, cell: CellCoordinates, val: int, overwrite: bool = False):...

pierluigimarchioro
def empty_cell(self, cell: CellCoordinates):...

pierluigimarchioro
def get_row(self, i: int) -> np.ndarray:...

pierluigimarchioro
def get_column(self, i: int) -> np.ndarray:...

pierluigimarchioro
def get_square(self, cell: CellCoordinates) -> np.ndarray:...

pierluigimarchioro
def is_full(self) -> bool:...
```

Figure 2.3: *SudokuGrid* Class (2)

2.2.2 Main Function

This part is the actual implementation of the Backtracking Sudoku-solving function. The main idea is of course to try to assign some value to some cell and then move on to the next; if a dead end is reached, the algorithm retraces back to the first cell with available assignments and tries again to move forward. As regards to this particular implementation, which makes use of *Constraint Propagation*, the main steps of the algorithm are the following:

1. **Choice of the Cell to Examine**: each iteration of the algorithm examines the most constrained cell at that point, that is, the cell whose domain is the smallest; this dynamic ordering of the variables was chosen because of the intuition that at least one value in each domain must be the correct one, so it's better to start from the cells with the fewest choices.

2. **Backtracking**: the algorithm backtracks whenever a cell that has an empty domain is encountered, when every value in the domain of the current cell has been attempted, or when the grid is full but it isn't a valid solution.

3. **Domain Updates**: during each move forward or backward of the algorithm, the domains of the cells affected by some new assignment are updated accordingly. Practically, this happens inside the *grid.set_value* and *grid.empty_cell* method calls, however such a process is discussed more in depth in the following sections.

```python
# If None it means that all cells have been exhausted and the sudoku was solved
if current_cell is None:
    return ut.is_solution_correct(grid), grid

if len(current_domain) == 0:
    # Empty domain means the solver reached a dead end, hence return False
    return False, grid

for attempt in current_domain:
    # Allow current cell to be overwritten
    grid.set_value(cell=current_cell, val=attempt, overwrite=False)

    # Go to the next cell after having tried with the current attempt
    next_cell, next_domain = grid.get_minimum_domain_empty_cell()
    solved, solution = solver_aux(
        grid=grid,
        current_cell=next_cell,
        current_domain=next_domain
    )

    if solved:
        return True, solution

# If no solution found at current cell, set as empty again and go back
grid.empty_cell(cell=current_cell)
return False, grid
```

Figure 2.4: Main Body of the recursive BT algorithm - The domain of each empty cell is iterated through, starting from the most constrained cells

2.2.3 Forward Checking and Domain Updates

The actual *Constraint Propagation* aspect of this solver is implemented through a form of *Forward Checking*: after each assignment to some cell C, which can be both the action of setting a set to a new value and the action of emptying the cell, all the cells affected by it, that is, the cells that share the same row, column or box of C, have their domain updated to reflect only the values that can be assigned to them, in the future, without violating any constraint.

More implementative details are discussed in the following sub-sections.

2.2.3.1 Implementation of Cell Domains

Cell domains were implemented as a dictionary that pairs each cell (coordinates) with its set of valid assignments, which are, of course, numbers from 1 to 9. Such a dictionary is a field of the *ConstraintPropagationSudokuGrid* class, which, as the name says and as previously mentioned, implements constraint propagation functionality.

```
pierluigimarchioro
class ConstraintPropagationSudokuGrid(SudokuGrid):

    _cell_domains: Dict[CellCoordinates, Set[int]]
    """
    Dictionary that pairs each cell coordinate with its domain, which consists
    in the set of 1 to 9 integers that can be assigned to that cell without
    violating any constraint
    """
```

Figure 2.5: Implementation of cell domains - Python dictionary that pairs each cell (coordinates) with its set of valid assignments

2.2.3.2 Implementation of Domain Updates on New Assignments

The basic operations to update the domains of cells affected by new assignments are, in fact, the calculation of the domain of a single cell, and the calculation of the set of cells affected by some assignment.

Cell Domain Calculation The domain of a cell C is defined as the set of values that can be assigned to it without violating any constraint, which means that such values must necessarily not occur in the row, column and box that C belongs to. Such a set is obtained as the intersection of the sets of values not contained in such collections of cells.

```
pierluigimarchioro
def _calculate_cell_domain(self, cell: CellCoordinates) -> Set[int]:
    # All available numbers (moves)
    maximum_domain = set(range(1, 9+1))

    row_domain = set(self.get_row(i=cell.row))
    available_on_row = maximum_domain.difference(row_domain)

    col_domain = set(self.get_column(i=cell.col))
    available_on_col = maximum_domain.difference(col_domain)

    square_domain = set(self.get_square(cell=cell).flatten())
    available_on_square = maximum_domain.difference(square_domain)

    # Cell domain = intersection of available numbers in its row/col/square
    return set.intersection(available_on_row, available_on_square, available_on_col)
```

Figure 2.6: Calculation of the domain of a single cell

Affected Cells Calculation The set of cells affected by a new assignment to some cell C is simply calculated as the set of coordinates belonging to the cells that share the same row, column or box as C.

```
pierluigimarchioro
def _get_affected_cells(self, cell_to_update: CellCoordinates) -> Set[CellCoordinates]:
    # Check if the set of affected cells has been calculated before
    if cell_to_update in self._affected_cells_cache:
        return self._affected_cells_cache[cell_to_update]
    else:
        row_cells = {
            CellCoordinates(row=cell_to_update.row, col=i)
            for i in range(0, 8+1)
        }
        col_cells = {
            CellCoordinates(row=i, col=cell_to_update.col)
            for i in range(0, 8+1)
        }

        square_starting_row = (cell_to_update.row // 3) * 3
        square_starting_col = (cell_to_update.col // 3) * 3
        square_cells = {
            CellCoordinates(row=i, col=j)
            for i in range(square_starting_row, square_starting_row+3)
            for j in range(square_starting_col, square_starting_col+3)
        }

        affected_cells = set.union(row_cells, col_cells, square_cells)
        self._affected_cells_cache[cell_to_update] = affected_cells

        return affected_cells
```

Figure 2.7: Calculation of the set of cells affected by the assignment of a new value to some cell C - It is noted that the results of each call are cached, because the set of affected cells of C never changes

As regards to the actual updates to the domains, they are performed in different manners, depending on the type of the new assignment.

Emptying a Cell In such a case, the domains of all the affected cells are recalculated entirely, because it is not certain that all of the former would expand after removing one number from the grid; if that was the case, then it would've been enough to just add the removed value back to them.

Filling a Cell In such a case, the new value is simply removed, if present, from the domains of the affected cells. This is a simple condition to check, and it is far more efficient than recalculating domains in their entirety.

```
pierluigimarchioro *
def _remove_from_affected_domains(self, val: int, cell_to_update: CellCoordinates):
    affected_cells = self._get_affected_cells(cell_to_update=cell_to_update)
    for c in affected_cells:
        if val in self._cell_domains[c]:
            self._cell_domains[c].remove(val)   You, 13/11/22, 17:29 • Uncommitted c

pierluigimarchioro
def _recalculate_affected_domains(self, cell_to_update: CellCoordinates):
    affected_cells = self._get_affected_cells(cell_to_update=cell_to_update)
    for c in affected_cells:
        new_domain = self._calculate_cell_domain(cell=c)
        self._cell_domains[c] = new_domain
```

Figure 2.8: Update of the domains of the cells affected by a new assignment

2.2.4 Ordering Technique

As discussed before, the algorithm leverages a dynamic ordering of the variables, that is, the Sudoku cells, which is based on the dimension of their domains; specifically, the most constrained empty cell is chosen first. Below is the implementation of this ordering mechanism.

```
pierluigimarchioro *
def get_minimum_domain_empty_cell(self) -> Tuple[CellCoordinates, Set[int]] | Tuple[None, None]:
    """
    Returns the coordinates of the empty cell with the minimum (smallest) domain,
    along with said domain.

    :return: cell with smallest non-empty domain and its domain,
        or the pair (None, None) if there are no empty cells left
    """

    # Get empty cells
    empty_cells = list(
        filter(
            lambda kv_pair: self.get_value(kv_pair[0]) == self.empty_cell_marker,
            self._cell_domains.items()
        )
    )

    if len(empty_cells) == 0:
        return None, None

    # Get the empty cell with the smallest domain
    min_cell, min_domain = min(empty_cells, key=lambda kv_pair: len(kv_pair[1]))

    return min_cell, copy(min_domain)
```

Figure 2.9: Implementation of dynamic ordering - the most constrained cell is chosen first

3 Simulated Annealing

3.1 Theoretical Foundations

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function, often used when the search space is discrete. The name of the algorithm comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties. Simulated annealing can be used for very hard computational optimization problems where exact algorithms fail; even though it usually achieves an approximate solution to the global minimum, it could be enough for many practical problems. This notion of **slow, controlled cooling** implemented in the simulated annealing algorithm **is interpreted as a slow decrease in the probability of accepting worse solutions** as the solution space is explored. Accepting worse solutions allows for a more extensive search for the global optimal solution, because it allows to overcome the characteristic shortcoming of simple hill-climbing algorithms, that is, being very prone to find local minima.

3.1.1 Fundamental Concepts

In general, simulated annealing algorithms work as follows. The temperature progressively decreases from an initial positive value to zero. At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and moves to it according to the temperature-dependent probabilities of selecting better or worse solutions, which during the search respectively remain at 1 (or positive) and decrease towards zero.

Each fundamental concept of the Simulated Annealing approach is better described in the following paragraphs.

Energy Function Energy is defined as a function that assigns a score to some provided state of the system. The better the state, that is, the closer such a state is to the solution of the problem, the lower the score assigned to it. Therefore, minimizing the energy function means trying to find a solution to the problem at hand.

The basic iteration At each step, the simulated annealing heuristic considers some neighboring state $s^\star$ of the current state s, and probabilistically decides between moving the system to state $s^\star$ or staying in the current state s. These probabilities ultimately lead the system to move to states of lower energy. Typically, this step is repeated until the system reaches a state that is good enough for the application, or until a given computation budget has been exhausted.

Neighbouring States The neighbours of a state of the problem are defined as states produced by applying minimal alterations to the former. Such states are evaluated when trying to optimize

the solution in an attempt to progressively improve the current state. Each neighbouring state.

Acceptance Probabilities The probability of making the transition from the current state s to a candidate new state s_{new} is specified by an acceptance function $P(e, e_{new}, T)$, that depends on the energies e and e_{new} of the two states, and on a global time-varying parameter T called temperature. It is recalled that states with a smaller energy are better than those with a greater energy. Therefore, in order to prevent the method from becoming stuck at a local minimum that is worse than the global one, the probability function P must be positive even when e_{new} is greater than e. However, when T tends to zero, the probability $P(e, e_{new}, T)$ must tend to zero if $e_{new} > e$ and to a positive value otherwise. For sufficiently small values of T, the system will then increasingly favor moves that go *downhill* (i.e., to lower energy values), and avoid those that go *uphill*. With $T = 0$ the procedure reduces to the greedy algorithm, which makes only the downhill transitions.

A commonly used acceptance probability function with the above characteristics is defined as

$$P(e, e_{new}, T) = \begin{cases} 1, \text{ if } e < e_{new} \\ P[X \leq \exp(\frac{-(e_{new}-e)}{T})] \text{ otherwise} \end{cases}$$

where X is a random variable that follows a uniform distribution in the closed interval $[0, 1]$.

3.1.2 Theory Applied to the Sudoku Problem

The solution of the Sudoku problem is a grid that completes the partially initialized, starting one, with 1 to 9 digits, in such a way that there are no repetitions of numbers in every row, column and box. For this reason, the energy function can be defined as one that penalizes states with a higher number of repetitions across the three Sudoku "dimensions" (col, row, box), i.e. states with a higher number of repetitions have a higher energy. Additionally, in Sudoku, a neighbouring state can be thought of as a Sudoku grid that differs from the original state by just a handful of cells, possibly just one or two. Finally, the probability acceptance function can be the commonly used one defined in the previous section.

3.2 Software Implementation

The core parts of the implementation are the main function, which implements the simulated annealing process, state representation and the way neighbors are generated, the energy function, the acceptance probability function and the temperature reset process in case the solver is stuck on a local minimum.

3.2.1 Fundamental Data Structures

The majority of the fundamental data structures used to implement this approach are the same ones described at section 2.2.1. More specifically, *SudokuGrid* serves as the supertype of the class *SimulatedAnnealingGrid*, which basically represents the state of the system. As such, it provides an interface to easily calculate its energy and to generate a neighbouring state, i.e. perform the so called *move* action.

3.2.2 Main Function and General Intuition

The general intuition behind the procedure used to find the Sudoku solution is that, at each iteration of the algorithm (referred to as *epoch*), the next (neighboring) state is obtained by swapping two cells of a same, random, row, and this is repeated until a solution is found or some computational budget is exceeded. During the initialization phase, in fact, the starting grid was filled in such a way that each row contained exactly one occurrence of every Sudoku number (from 1 to 9). This approach allowed to reformulate the problem into finding the correct set of row permutations such that all the Sudoku constraints are respected.

Below is an image of the main part of the aforementioned Simulated Annealing loop.

```
solution = None
epoch = 0
while solution is None and epoch <= params.max_epochs:
    new_state = current_state.get_next()
    score_delta = new_state.score - current_state.score
    if score_delta <= 0:  # new state is better
        current_state = new_state
    else:  # old state is better (delta > 0)
        # The right side of the inequality gets closer to 0 the lower the temperature
        if np.random.rand() < 10 ** (-score_delta / current_state.temperature):
            current_state = new_state

    # Terminate if solution found (score == 0)
    if current_state.score == 0:
        solution = current_state.grid
        break

    current_state.temperature *= params.temp_decrease_rate
    epoch += 1
```

Figure 3.1: Main Simulated Annealing loop

Another important part of the development of the solver was to introduce a *temperature reset step* in case the algorithm stayed "stuck" for too long. This, along with the parameters of the solver and a brief parenthesis on the acceptance probability function, are better discussed in the following subsections.

3.2.2.1 Parameters

The solver accepts the three main parameters shown, along with their respective descriptions, in the below image. The initial values assigned to these parameters were empirically determined:

- a *starting temperature* of 3, which is a value close to that of T such that $\log_{10} -\Delta E/T = 0.7$, with $ExpectedValue(\Delta E) \sim 2$; in other words, the starting temperature was chosen so that the probability of accepting a worse state at the beginning was close to 70%
- a *temperature decrease rate* of 0.95, with the goal of decreasing the temperature relatively fast, albeit not too much; this was to compensate for the temperature reset step, that periodically raised the temperature to a relatively high level

- a *computational budget*, represented by the *max_epochs* parameter, big enough to allow to solve even relatively hard Sudoku grids; the value of such a parameter was set to $3 * 10^6$

```
pierluigimarchioro
@dataclass
class SimulatedAnnealingParams:
    starting_temp: float
    """
    Starting temperature value used in the annealing approach
    """

    temp_decrease_rate: float
    """
    Rate, less than 1, that the temperature decreases at.
    i.e. T_X+1 = T_X * rate
    """

    max_epochs: int
    """
    Max number of epochs to execute the solver for
    """
```

Figure 3.2: Parameters of the Simulated Annealing procedure

3.2.2.2 Acceptance Probability Function

The implementation of such a function can be seen at image 3.1, and it refers to the "canonical" acceptance probability function described at section 3.1.1.

3.2.2.3 Temperature Reset

As mentioned before, in order to allow the algorithm to "recover" in case it was "stuck", a temperature reset step was implemented. *Being stuck* means that the score of the state chosen at each iteration didn't vary for more than *stuckness_threshold* times, with *stuckness_threshold* being set to 100. For each iteration at and after such a threshold was reached, the formula in the following image was applied to obtain the new temperature level:

```
    # Temp reset is lower the more time passes
    epoch_multiplier = max((1 - (epoch / params.max_epochs)), 0.01)
    if prev_counter > stuckness_threshold:
        current_state.temperature = min(
            params.starting_temp * epoch_multiplier * stuckness_multiplier,
            params.starting_temp
        )
```

Figure 3.3: Temperature Reset Formula

It is noted that the new temperature reset level is a function of the *starting temperature*, as well as of two other parameters:

- *stuckness_multiplier*, which increases the more the algorithm is stuck on a certain score
- *epoch_multiplier*, which decreases the more the computational budget of the algorithm is exhausted (i.e. time passes); additionally, this parameter was capped at 0.01 so that the probability of accepting a worse state would never become too low

3.2.3 State Representation and Neighbor Generation

State representation and neighbor generation were both mainly handled by the *SimulatedAnnealingState* class, which provided as a useful wrapper for a, previously described, *SimulatedAnnealingGrid* instance. The goal of such a class was to provide a simple interface to perform the *move* action, implemented by the *get_next()* method, and the calculation of the energy of some grid state, referred to as *score*.

The below images show how such an interface and such actions were implemented.

```
pierluigimarchioro
@dataclass
class SimulatedAnnealingState:
    """
    Class that represents the state of an independent Simulated Annealing
    solving run
    """

    temperature: float
    grid: SimulatedAnnealingSudokuGrid
    _score: float = ut.LARGE_NUMBER

    pierluigimarchioro
    def __str__(self):...

    pierluigimarchioro
    @property
    def score(self) -> float:
        if self._score == ut.LARGE_NUMBER:
            self._score = self.grid.get_score()

        return self._score

    pierluigimarchioro
    def get_next(self) -> SimulatedAnnealingState:
        # Important to copy, else the state pre- and post- update
        #   is virtually the same
        new_grid = deepcopy(self.grid)
        new_grid.swap_two_cells_same_row()

        return SimulatedAnnealingState(
            temperature=self.temperature,
            grid=new_grid
        )
```

Figure 3.4: *SimulatedAnnealingState* class

```
pierluigimarchioro *
def swap_two_cells_same_row(self) -> Tuple[CellCoordinates, CellCoordinates]:
    """
    Swaps two cells that belong to the same row.
    :return: coordinates of the two swapped cells, which cannot refer
        to cells that were already filled in teh starting grid
    """

    cell_1, cell_2 = self._get_random_row_neighbors()
    val_1, val_2 = self.get_value(cell_1), self.get_value(cell_2)

    self.set_value(cell_1, val_2)
    self.set_value(cell_2, val_1)

    return cell_1, cell_2
```

Figure 3.5: Implementation of the *move* action - Two random cells of the same random row are swapped

3.2.4 Energy Function

As hinted to before, this is referred to, in the solver implementation, as *scoring function*. Such a function was defined so that the score associated to a Sudoku grid was the sum of the duplicates in each of its columns and boxes. Rows were not counted, because, as previously discussed at 3.2.2, the solver maintains the invariant that each row always contains exactly one occurrence of each Sudoku number, therefore never presenting any duplicate.

```
pierluigimarchioro *
def get_score(self) -> float:
    """
    Returns the score associated to the grid, also called `energy`
    in the simulated annealing context.
    :return: score (energy) of the current grid
    """

    pierluigimarchioro
    def get_collection_score(collection: np.ndarray) -> float:
        duplicates, counts = ut.get_duplicates(collection)
        duplicates_sum = int(np.sum(counts)) if len(counts) > 0 else 0

        return duplicates_sum

    tot_score = 0
    for i in range(0, 8+1):
        col_score = get_collection_score(self.get_column(i))
        tot_score += col_score

    for i in range(0, 2+1):
        for j in range(0, 2+1):
            top_left_cell = CellCoordinates(row=i*3, col=j*3)
            square_score = get_collection_score(self.get_square(top_left_cell))
            tot_score += square_score

    return tot_score
```

Figure 3.6: Implementation of the energy (scoring) function

Final Considerations

The two approaches implemented for this assignment are very different in nature: *Constraint Propagation* is based on the exploitation of a set of well defined rules to reduce the search space and greedily come to the final solution, while *Simulated Annealing* is a stochastic process that tries to optimize some type of score, the energy, by merely performing hill-climbing, with the added characteristic of being able to overcome local minima worse than the global one. As such, the former approach is the best suited to solve the Sudoku problem, since the set of constraints is simple and well defined. The latter, instead, while being almost always able to find a solution given a reasonable computational budget, tends to suffer, at least when compared to the Constraint Propagation approach, because of the combinatorial aspect of the problem, which makes the search space big and hence relatively difficult to explore in a stochastic manner.

Empirical testing has shown that the *Backtracking, Constraint Propagation* solver always outperforms the *Simulated Annealing* one, thereby proving that the latter is ill-suited to solve the Sudoku problem.

List of Figures