Università
Ca' Foscari
Venezia

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

Solving a sudoku with Back-tracking and Constraint Propagation and Simulated Annealing

Studentessa
Serena Zanon
Matricola 887050

Anno Accademico
2023 / 2024

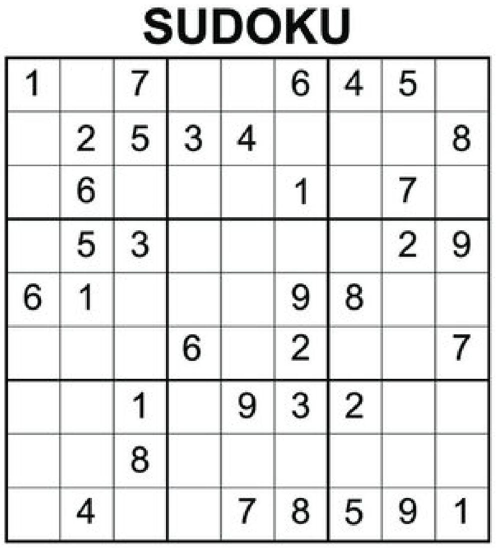
Indice

Introduction: What is a Sudoku?

A Sudoku puzzle is a logical game which exploits the combinatorial placement of numbers. This game consists in filling a (9x9) grid with digits from 1 to 9 such that 3 simple constraints are respected:

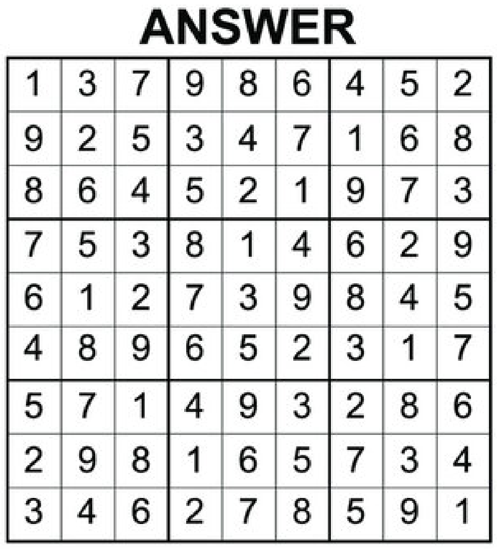
- Each digit has to appear in each row;
- Each digit has to appear in each column:
- Each digit has to appear in each of the 9 (3x3) blocks that compose the grid.

The initial state of the grid presents some of the cells with a digit which cannot be modified and according to them the player has to guess the right spot for every missing digit. [6]



SUDOKU

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



ANSWER

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

The assignment

The assignment proposed asks to write two algorithms that solve a whatever Sudoku, compare them through some experiments and declare which is the best.
Following here is the assignment request:
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:

1. Simulated annealing;
2. Genetic algorithms;
3. 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 a known square should be represented by the corresponding digit (1,...,9).
For example:

```
37.5....6
...36..12
....9175.
...154.7.
..3.7.6..
.5.638...
.6498....
```

```
59..26...
2....5.64
```

The proposed Sudoku solvers will take in input a *.txt* file containing some Sudoku with different levels of difficulty (in order 5 easy, 5 normal, 5 medium, 10 hard) and each Sudoku is represented by a line of digits where '0' corresponds to an empty Cell.
For example:

```
370500006000360012000091750000154070003070600050638000064980000590026000200005064
```

Therefore, in the code, there's a function that extracts randomly a Sudoku, format it in a matrix shape in which each row has 9 digits and solves it.

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

The solvers use Python 3.12-64-bit along with some defined libraries: it is necessary to install these libraries in order to run the code.
These libraries can be installed using the command *"pip3 install library_name"* in the command prompt of the device in which this code is going to be runned.

Solving Sudoku with a constraint satisfaction approach

3.1 Constraint satisfaction problem

A constraint satisfaction problem (CSP) is defined by three sets:

- $X = \{X_1, \cdots, X_n\}$ is the set of variables;
- $D = \{D_1, \cdots, D_n\}$ is the set of domains and each D_i is associated with the corresponding variable X_i;
- $C = \{C_1, \cdots, C_n\}$ is the set of constraints.

Each domain D_i contains allowable values for the variable X_i and each constraint C_i is given by a pair $<$*scope, rel*$>$: [3]

- *scope* is a subset of X, aka a set of variables;
- *rel* specifies the relation on the allowable values that can be assigned to variables contained in *scope*, so it is a subset of D.

Solving CSP involves some combination of: [4]

- Constraint propagation;
- Search to explore valid assignments.

For constraint propagation is intended the process of communicating the domain reduction of a variable to all the constraints stated over this variable.

This process is considered finished when it's no longer possible to reduce a variable's domain or when the domain is empty and the execution of the algorithm fails.

Although, the constraint propagation does not guarantee to find the final solution: this could happen when constraints are weak or when there are too many constraints to satisfy.

In these cases, it's essential to search in the state-space through backtracking.

In order to define the term *backtracking*, some definitions has to be declared:

- An assignment is considered consistent, if it does not violate any constraint;
- An assignment is considered complete, if every single variable of the set X is assigned;
- A solution is a complete and consistent assignment;
- An assignment is considered partial, if only a few variables are assigned (obviously, this assignments must not violate any constraints). [3]

Backtracking identifies a class of algorithms in which candidates to the solution are built incrementally and consecutively abandoned if they cannot become a valid solution to the problem.

In other words, the state-space can be imagined as a tree in which the root is the initial state of variables and each node of the layers below represents a partial assignment: if some assignments are inconsistent, their corresponding nodes are removed from the tree and only those representing consistent assignments are visited. [7]

3.2 Sudoku as constraint satisfaction problem

The Sudoku game can be reduced to a constraint satisfaction problem and can be defined as follows:

- $X = \{X_1, \cdots, X_{81}\}$ is the set of variables, precisely the 81 cells composing the grid;
- $D = \{D_1, \cdots, D_{81}\}$ is the set of domains and each domain could contain digits from 1 to 9 according to the correspondent variable:
- C can be the set of direct constraints or indirect constraint:
  - $C = \{C_1, \cdots, C_{27}\}$ is the set of direct constraints, where:
    - $\{C_1, \cdots, C_9\}$ are called "Row Constraints" meaning that each value in the same row must be different from the others;
    - $\{C_{10}, \cdots, C_{18}\}$ are called "Column Constraints" meaning that each value in the same column must be different from the others;
    - $\{C_{19}, \cdots, C_{27}\}$ are called "Square Constraints" meaning that each value in the same (3x3) square must be different from the others.
  - $C = \{C_1, \cdots, C_9\}$ is the set of indirect constraints and each C_i is related to the i^{th} digit meaning that the number *i* must appear in a single row, column and Sudoku square only once.

3.3 Implementation of constraint satisfaction and backtracking algorithm to solving Sudoku

In this chapter, there is the explanation of the code implemented and the logic behind it.
The program calls just the following function:

```
def main():
```

In this function are called two principal functions:

- *set_up_sudoku*

  ```
          def set_up_sudoku()
                              -> list[list]:
  ```

  It has no input parameters and its role is to randomly extract from a file *.txt* a Sudoku. This file contains 25 Sudoku with different levels of difficulty: in this way the algorithm can be tested on different Sudoku puzzles each time it is computed.

- *solution*

  ```
          def solution(board: list[list], row: int, column: int)
                                                                  -> bool:
  ```

  This function is the principal function and it contains the whole procedure to reach the final result. It accepts in input the Sudoku's board and two int values (corresponding to the row and to the column of a cell) and returns a boolean value.

Now, here is the complete explanation of the function *solution*.
As already mentioned, *solution* has as input parameters the board and the coordinates of a cell: when this function is called in the *main* function,

the first parameter is equal to the result of *set_up_sudoku* function, while the second parameter is equal to the coordinates of the first left cell, so (0, 0).

Then there are some base cases:

- If the value of the row is equal to 9 then the algorithm has reached the bottom of the board and the value *True* is returned since there are no more cells to visit;

- If the visited cell has a value different from '0', which means that it's a cell containing a value between 1 and 9, the next cell's coordinates are calculated and the function *solution* is called.

Otherwise, if an empty cell is reached (its content is equal to '0') the algorithm does the following:

- Calculates the domain of the visited cell with the function *calcula-te_domain*

  ```
  def calculate_domain(board: list[list], row: int, column: int)
                                                                   -> list:
  ```

  This function takes in input the board and the cell, in order to calculate its domain. In this way, only the possible values that this particular cell can assume are checked. To do so are called other functions:

  ```
  def values_row(board: list[list], row: int)
                                             -> list:

  def values_column(board: list[list], column: int)
                                                   -> list:

  def values_square(board: list[list], row: int, column: int)
                                                             -> list:
  ```

The function *values_row* returns a list containing all the values already set in that particular row. The same is done by *values_column* and *values_square* respectively on that particular column and on that particular (3x3) square. Then the function *reverse_domain* is called on all three lists just obtained

```
def reverse_domain(values: list)
                                                -> list:
```

Its aim is to return a list with the missing values from 1 to 9 of the input list.

As last step, *calculate_domain* does the intersection between the three reversed lists obtained and returns it as the domain of the visited cell.

- It's necessary, for each value in the domain just computed, to control if its assignment to the variable does not violate any constraint. In order to do that, the function *check_constraints* was implemented.

  ```
  def check_constraints(board: list[list], row: int,
                                        column: int, val: str) -> bool:
  ```

  check_constraints takes as input the board and the coordinates of the visited cell and returns *True* if all constraints are respected and *False* otherwise: if only one constraint is violated then this function gives *False* as result.

  To check the three main constraints of the game are called three different functions:

  ```
  def check_row(board: list[list], row: int, val: str)
                                                -> bool:

  def cehck_column(board: list[list], column: int, val: str)
                                                -> bool:
  ```

```
def check_square(board: list[list], row: int, column: int,
                                              val: str) -> bool:
```

Each of these functions returns *True* if the value of the parameter *val* is not in that particular row, column or square.

- If *check_constraints* is successfully passed then
  - The assignment of the controlled value to the visited cell is done;
  - The next position is calculated and *solution* is called on it

  If *solution* returns *True* then it means that this branch of the tree could become a valid solution, so a positive response is given. If *solution* does not return a positive result then another value is checked and so on until all the allowable values are checked.

- After the *check_constraints' if*, the value of the visited cell returns to being a '0', in order to give the possibility to find other possible solutions to the game.

- If neither of the values in the domain can be assigned without violating any constraints then a negative response is returned.

Solving Sudoku with a simulated annealing approach

4.1 Simulated annealing

Simulated annealing (SA) is a metaheuristic optimization technique for approximating the global optimum of a given function *f*, specifically it is widely used to solve optimization problems characterised by a large search space in order to approximate the global optimization.

The name's algorithm is based on the metallurgy's annealing process, which is a technique where a metal is heated to a high temperature very quickly and then it is cooled gradually obtaining an alteration of its physical properties. [8]

Therefore, the algorithm itself tries to emulate this particular process: the initial state is considered as a high-energy state and slowly the algorithm lowers the temperature, which is a parameter, until the minimum-energy state is reached. [1]

To do so, a neighbouring state of the current scenario is considered and with a certain probability the algorithm chooses either to remain in the current state or to move to the new one.

This probability depends on the energies of the interested states and the temperature parameter, which changes during the execution of the algorithm.

- If the temperature is low, then the probability of accepting the new state is low;
- If the temperature is high then the probability of accepting the new state is high.

Similarly to the metallurgical process, the temperature is lowered and if this parameter is decreased slowly enough, then the search of the solution will reach a global optimum with probability approaching to 1. [5]

One of the major problems of this algorithm is to choose the right values for the initial temperature and the cooling rate.

As already mentioned, the temperature during the execution of the algorithm changes: at the beginning of the computation, it is necessary that the probability of accepting a new state is high and, in the farther iterations, gradually reduced: in this way, the algorithm becomes greedier and greedier and the choice will fall on the best-case scenario.

One possible way to calculate the initial temperature is measuring the variance in cost for a small neighbourhood of the initial state.

Regarding the cooling rate, one possible method to compute the lowering of the temperature is using the geometric cooling schedule: the new temperature (t_{i+1}) is obtained as a function of the current temperature (t_i) as follows:

$$t_{i+1} = \alpha * t_i$$

Now the question is about which value can assume α, that is the cooling rate: this value has to be contained in the interval (0, 1) and for large values, like 0.99, the temperature will decrease very slowly, which is the desired situation. [2]

4.2 Implementation of simulated annealing algorithm to solving Sudoku

In this chapter, there is the explanation of the code implemented and the logic behind it.
The program calls just the following function:

```
def main():
```

In this function are called two principal functions:

- *set_up_sudoku*

  ```
  def set_up_sudoku()
                      -> list:
  ```

  As said in chapter 3, it has no input parameters and its role is to randomly extract from a file *.txt* a Sudoku puzzle.

- *solution*

  ```
  def solution(board: list)
                      -> list:
  ```

  This function is the principal function and it contains the whole steps of the simulated annealing algorithm. It accepts in input the Sudoku's board and returns the final board.

Following, there is the extended explanation of the *solution* function.

- The very first rows of the function are aimed to initialise some essential variables for the algorithm:
  - A boolean variable (*victory*) which tells if a solution has been found;
  - The cooling rate, whose value is set to 0.99 (according to the reasons written in the previous chapter);

  - A list given by 9 lists, each of them indicating the positions of each block composing the grid;
  - A list containing the positions of the fixed cells.

- Then the initial board is randomly filled by the *fill_board* function and the initial temperature and the initial error are computed.

```
def fill_board(board: list):
```

It accepts as input the board and returns nothing. Its goal is to assign to the empty cells (the ones with value equal to '0') a random number in such a way that this number is unique in the (3x3) square to which the cell belongs. The initial temperature is calculated thanks to the function *calculating_temperature*:

```
def calculating_temperature(board: list, blocks: list[list],
                            fixed_cells: list[list]) -> float:
```

As announced in the above chapter, the temperature is typically obtained by considering the variance of the cost of a small neighbourhood of the initial state. To implement such a procedure, 10 different costs of different possible states derived from the initial one are calculated and then their variance is retrieved.

  - To perform the different possible states *new_version_sudoku* function is called:

    ```
    def new_version_sudoku(board: list, blocks: list[list],
                           fixed_cells: list[list]) -> list:
    ```

    The aim of this function is to create a new version of the board from the one passed in input: a random block is chosen and on that block two no-fixed random cells are chosen and then swapped. The definitions of the auxiliary functions called in this scope are:

```
def random_cells(block: list, fixed_cell: list[list])
                                          -> list[list]:
```

This function finds, in the given block, two no-fixed cells and returns their coordinates.

```
def switch(board: list, first_pos: list,
                            second_pos: list) -> list:
```

The only thing that this function does is swapping the values of the *first_pos* cell and the *second_pos* cell and returning the modified board.

- After calculating a new possible version, it's computed its cost by the function *cost_function*:

```
def cost_function(board: list)
                            -> int:
```

Its returned result is given by the sum of two other functions' results: *row_cost* and *column_cost*:

```
def cost_row(board: list)
                  -> int:
```

cost_row computes the cost of each row and returns their sum. A cost of a single row is given by of how many numbers are not unique in that specific row.

```
def cost_column(board: list)
                  -> int:
```

cost_column computes the cost of each column and returns their sum. A cost of a single column is given by of how many numbers are not unique in that specific column.

- All the costs of all the ten versions are inserted in a list and their variance is returned as the initial temperature.

Trivially, the initial error is equal to the tenth version's error and obtained with the function *cost_function*.

- Since simulated annealing algorithm is random, it might be possible that a solution has already been reached so there is a control on the number of errors: if it is less or equal than zero, then the variable *victory* is set to *True*.

- After this control, there is the core of the *solution* function: until a solution is not found, the algorithm iterates for a specific number of iterations (which is set to the number of empty cells of the initial state due to the fact that the random numbers inserted are exactly equal to the number of the empty cells) and the function *choose_new_version* is called.

```
def choose_new_version(board: list, blocks: list[list],
                fixed_cells: list[list], temperature: float) -> [list, int]:
```

  The task of this function is to choose between the current state and one of its related neighbouring state, which is computed thanks to the function *new_version_sudoku*. Then their cost is obtained with *cost_function* and the difference between the cost of the new version and the old one is assigned to a variable called *delta*. This variable is fundamental in order to choose if the new Sudoku version is selected or not. The returned value is a pair in which the first element is the chosen board and the second is *delta* if the chosen board is the new version or 0 otherwise. The error produced is summed with the one already present and the same control about the number of errors is made.

- After all the iterations, the temperature is lowered and other checks are made:

- Since it has been kept in memory the number of errors of old versions, if the new error is bigger than the old ones then a counter called *stuck* is incremented;
- If this counter reached a number equal to 100 (variable *MAX_ITER*) then the temperature is increased so the algorithm is more likely accept new versions.

Experiments and comparison between the two adopted approaches

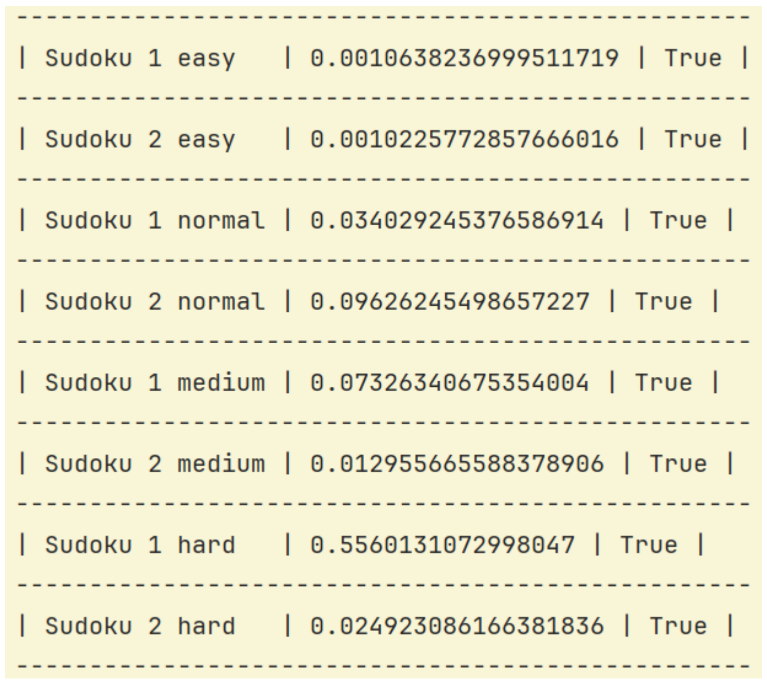
To evaluate and examine both approaches, an experimental set has been used.

This set is composed of exactly 8 Sudoku with 4 different levels of difficulty:

- 2 easy Sudoku;
- 2 normal Sudoku;
- 2 medium Sudoku;
- 2 hard Sudoku.

Following there are the tables containing the execution time of each Sudoku in both methods and if they found a solution for the puzzle.

- Backtracking and constraint satisfaction algorithm

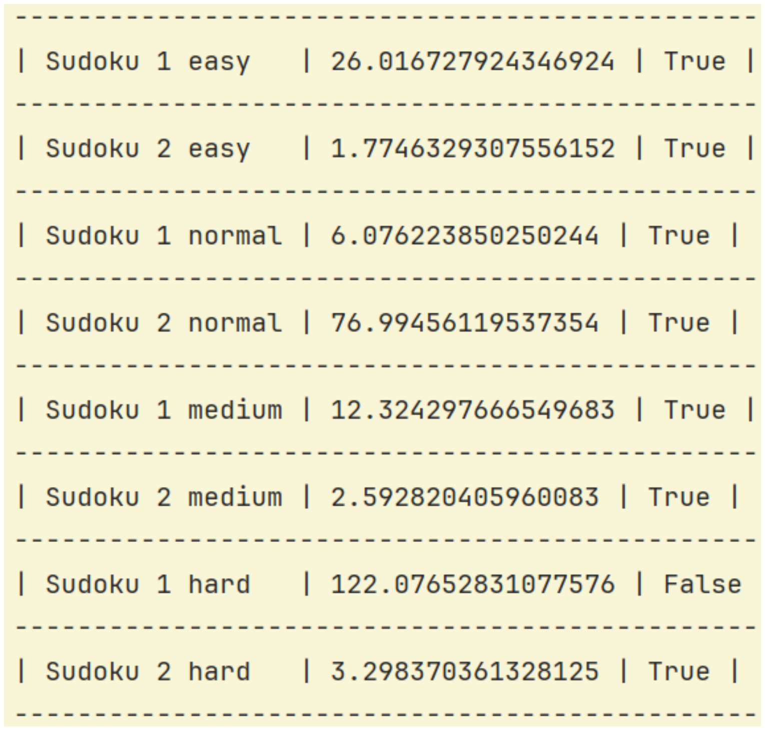


```
---------------------------------------------------
| Sudoku 1 easy   | 0.0010638236999511719 | True |
---------------------------------------------------
| Sudoku 2 easy   | 0.0010225772857666016 | True |
---------------------------------------------------
| Sudoku 1 normal | 0.034029245376586914 | True |
---------------------------------------------------
| Sudoku 2 normal | 0.09626245498657227 | True |
---------------------------------------------------
| Sudoku 1 medium | 0.07326340675354004 | True |
---------------------------------------------------
| Sudoku 2 medium | 0.012955665588378906 | True |
---------------------------------------------------
| Sudoku 1 hard   | 0.5560131072998047 | True |
---------------------------------------------------
| Sudoku 2 hard   | 0.024923086166381836 | True |
---------------------------------------------------
```

It is immediate to say that none of the 8 Sudoku took a lot of time to be computed and, most important thing, in every single one of them the algorithm has reached the global optimum, that is the solution of the board.

That's because the backtracking algorithm tries every single possible branch of the tree, ensuring that every possible and consistent node is visited and, at the same time, thanks to constraint propagation, some of those branches are dropped during the execution, decreasing the number of visits. One curious thing is that 'Sudoku 2 hard' took less time than some other Sudoku which level is easier: the reason why this may have happened is the fact that in the hard Sudoku the algorithm might immediately captured the right branch reducing the number of possibilities for the remaining empty cells.

- Simulated annealing algorithm



```
---------------------------------------------
| Sudoku 1 easy   | 26.016727924346924 | True |
---------------------------------------------
| Sudoku 2 easy   | 1.7746329307556152 | True |
---------------------------------------------
| Sudoku 1 normal | 6.076223850250244 | True |
---------------------------------------------
| Sudoku 2 normal | 76.99456119537354 | True |
---------------------------------------------
| Sudoku 1 medium | 12.324297666549683 | True |
---------------------------------------------
| Sudoku 2 medium | 2.592820405960083 | True |
---------------------------------------------
| Sudoku 1 hard   | 122.07652831077576 | False
---------------------------------------------
| Sudoku 2 hard   | 3.298370361328125 | True |
---------------------------------------------
```

To computing these experiments, some changes to the *solution* function has been made, although some Sudoku could take some hours to finish the entire computation. It has been introduced the variable *n_iter*: this variable takes part in the condition of the loop *while* in *and* with the one already existed: the algorithm continues to follow the *while* scope untill either a solution has been found or since *n_iter* is smaller or equal than *MAX_ITER*. From the table above, it's obvious that, since the simulated annealing algorithm is basically completely random, the execution time is very high, even for the simpler Sudoku. Indeed, the choice of the block and the choice of the two cells to invert are random: it can take a lot of time to guess the right block and invert the right cells.

Especially, at the beginning of the algorithm is more probable that the new state is accepted and, as the computation goes on, the probability decrease so it might be possible that there are only two cells in a block to be swapped but, again, since the choice is completely random it might take a lot of time cause it is less probable to catch the right block in which there are the two no-correct cells.

To calculate the execution time an *execution* function was implemented:

```
def execution(board: list) -> [float, bool]:
```

This function exploits the special python library *time* to count the amount of time requested in order to compute the solution for the input Sudoku'd board.

Conclusions

Summing up, back-tracking and constraint propagation algorithm is definitely better than the simulated annealing algorithm because with the first one every good, possible and consistent branch of the tree in visited.
On the other hand the simulated annealing algorithm is mostly based on random choices: it's more difficult for it to find the global optimum, especially in a little time as demonstrate in the previous chapter (if a limit on the number of iterations on the board is set, the algorithm would finish without even having found a solution for the game).
Due to these motives, back-tracking and constraint propagation algorithm is preferable than the simulated annealing algorithm.

Bibliografia

[1] Baeldung. Simulated annealing explained. URL `https://www.baeldung.com/cs/simulated-annealing`.

[2] Rhyd Lewis. *Metaheuristics can solve sudoku puzzles*.

[3] Norvig Stuart. *Artificial Intelligence: A modern approach (third edition)*.

[4] Andrea Torsello. *Constraint Satisfaction*. .

[5] Andrea Torsello. *Local Search*. .

[6] Wikipedia. Sudoku, . URL `https://en.wikipedia.org/wiki/Sudoku`.

[7] Wikipedia. Backtracking, . URL `https://en.wikipedia.org/wiki/Backtracking`.

[8] Wikipedia. Simulated annealing, . URL `https://en.wikipedia.org/wiki/Simulated_annealing`.