EXERCITATION 1 ARTIFICIAL INTELLIGENCE

Summary

Sudoku definition:

Sudoku is a very popular puzzle game that is played by millions of people every day. In spite of that, it is a NP-Hard problem that can be very difficult to solve depending on the initial conditions of the board. In this paper, we will write a solver for sudoku puzzles using a constraint satisfaction approach based on constraint propagation and backtracking and by using a technique called simulated annealing.

The object of sudoku is simple: given an 9 × 9 grid divided into 3 × 3 distinct squares, the aim is to fill each cell so that the following three criteria are met:

1. Each row of cells contains the integers 1 through to 9 exactly once.

2. Each column of cells contains the integers 1 through to 9 exactly once

3. Each n × n square contains the integers 1 through to 9 exactly once

Constraint Satisfaction Problems

Definition

Constraint satisfaction is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem. It consists of the following three components:

- **Variables**: entities that can have multiple values assigned to them {X_1, …, X_n}.
- **Domains:** the set of values {D_1, …, D_n} that can be assigned to a variable.
- **Constraints**: conditions on the valid assignments of values to variables, in the context of the other assignments

In this paper we will write a solver for sudoku puzzles using a constraint satisfaction approach based on constraint propagation and backtracking.

Constraint Propagation: Using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.

Backtracking search: a form of depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign, is commonly used for solving CSPs.

Minimum-remaining-values (MRV) heuristic: The idea of choosing the variable with the fewest "legal" or "most constrained variable". If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately, avoiding pointless searches through other variables.

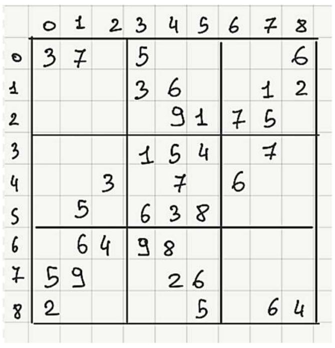
Forward Checking: Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.

Sudoku:

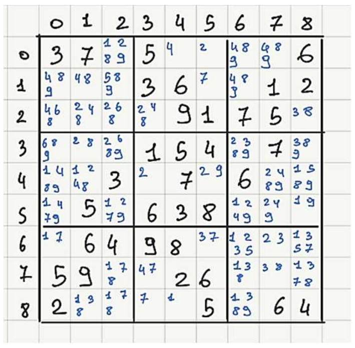
The three properties of the problem are defined as follow:

- Variables: Each empty cell on the board
- Domains: For each cell, a domain is defined as a set of numbers between 1 and 9 except the numbers which are already used in the current row, column or 3 x 3 squares.
- Constraints: The two types of constraints in the definition form as many types of constraints:
    - Direct constraints impose that no two equal digits appear for each row, column, and box;
    - Indirect constrains impose that each digit must appear in each row, column, and box.

The sudoku solver use *Constraint Propagation,* to maintain this feature we create a 9 x 9 matrix (called assign_matrix) which each cell(i ,j) corresponds to each variable(i ,j) of the sudoku. Inside each cell for the corresponding variable we have an array which contains only the values of its domain that respect the constraints. As we see in the following images, we fill the matrix with all elements in its domain but only that respects the constraints imposed (direct constraints and indirect constraints)

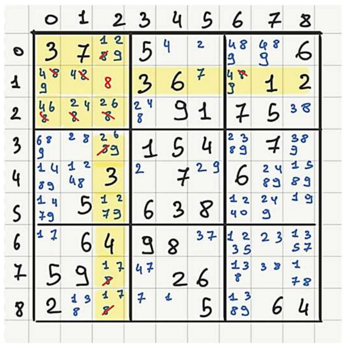


	0	1	2	3	4	5	6	7	8
0	3	7		5					6
1				3	6			1	2
2					9	1	7	5	
3				1	5	4		7	
4			3		7		6		
5		5		6	3	8			
6		6	4	9	8				
7	5	9			2	6			
8	2					5		6	4



	0	1	2	3	4	5	6	7	8
0	3	7	1 2 8 9	5	4	2	4 8 9	4 8 9	6
1	4 8 9	4 8	5 8 9	3	6	7	4 8 9	1	2
2	4 6 8	2 4 8	2 6 8	2 4 8	9	1	7	5	3 8
3	6 8 9	2 8	2 6 8 9	1	5	4	2 3 8 9	7	3 8 9
4	1 4 8 9	1 2 4 8	3	2	7	2 9	6	2 4 8 9	1 5 8 9
5	1 4 7 9	5	1 2 7 9	6	3	8	1 2 4 9	2 4 9	1 9
6	1 7	6	4	9	8	3 7	1 2 3 5	2 3	1 3 5 7
7	5	9	1 7 8	4 7	2	6	1 3 8	3 8	1 3 7 8
8	2	1 3 8	1 7 8	7	1	5	1 3 8 9	6	4

If we want to write a value on an empty cell in our sudoku in position (i, j), first we need to remove any occurrence of that value in the square 3x3 in which the cell(i ,j) is present. Then we have also to remove all occurrence of the value on the Row(X) and on Column(Y) where X = i and Y = j. As we see in the following images we want to set the cell(1, 2) = 8, to maintain the *Constraint Propagation* feature, we remove any occurrence of the value 8 in the square 3x3 where the cell is present, and in the Row(1) and Column(2). In this way we create a *new assign_matrix*, which each cell's domain respects the constraints imposed.



	0	1	2	3	4	5	6	7	8
0	3	7	1 2 ~~8~~ 9	5	4	2	4 8 9	4 8 9	6
1	4 ~~8~~ 9	4 ~~8~~	8	3	6	7	4 ~~8~~ 9	1	2
2	4 6 ~~8~~	2 4 ~~8~~	2 6 ~~8~~	2 4 8	9	1	7	5	3 8
3	6 8 9	2 8	2 6 ~~8~~ 9	1	5	4	2 3 8 9	7	3 8 9
4	1 4 8 9	1 2 4 8	3	2	7	2 9	6	2 4 8 9	1 5 8 9
5	1 4 7 9	5	1 2 7 9	6	3	8	1 2 4 9	2 4 9	1 9
6	1 7	6	4	9	8	3 7	1 2 3 5	2 3	1 3 5 7
7	5	9	1 7 ~~8~~	4 7	2	6	1 3 8	3 8	1 7 8
8	2	1 3 8	1 7 ~~8~~	7	1	5	1 3 8 9	6	4

Choosing the cell(1, 2), wasn't the best choice because we have its domain size is equal to 3. The best move is to first fill out the cells with smaller domains. For instance, if a cell's domain is [3] and another cell's domain is [1, 2, 9], it is obvious that filling the cell with a domain size of 1 is better since that's the only choice and it is definitely right.

So if you choose a value from the small domain set you have a high probability of choosing the right value. So we create a *list_of_variables*. The list is composed by dictionaries, which are composed by an "*index*" with the position on the matrix of the variable, and a "*len*" which contains the size of its domain. In this way we can sort our list and pop the first value, knowing that is the variable with the smallest domain size. In this way we fulfill the *Minimum-remaining-values* (MRV) heuristic.

First step: create the "*assign_matrix*" of the sudoku we want to solve. Then call the function "*solve_sudoku*" with parameters the sudoku (test0), the *assign_matrix* (assign) and an array where I keep track of the steps to solve the sudoku.

```
assign = create_assign_matrix(test0)

step = [0]

solve_sudoku(test0, assign, step)
```

Second step: after calling the function "*solve_sudoku*", we create the *list_of_variables*. If the list is empty this mean we reach our goal state, so we can terminate our recursion because our sudoku is solved.

If the *list_of_variables* is not empty, we sort it and pick the first value (*Minimum-remaining-values (MRV) heuristic*) and get our domain's value.

```
def solve_sudoku(sudoku, assign, step):
    #list with all variables (empty cells)
    list_of_var = create_list(sudoku, assign)

    #if we don't have any variables, we reach our goal state
    if len(list_of_var) == 0:
        return True

    #minimum remaining value heuristic
    #sort the list by len of the domain for each variable
    list_of_var.sort(key=order_by_len)
    #pop the first element of the list(element)
    index = list_of_var.pop(0).get('index')

    #domain of the variable
    domain = assign[index[0], index[1]]

    #number of recursive steps to solve the sudoku
    step[0] +=1
```

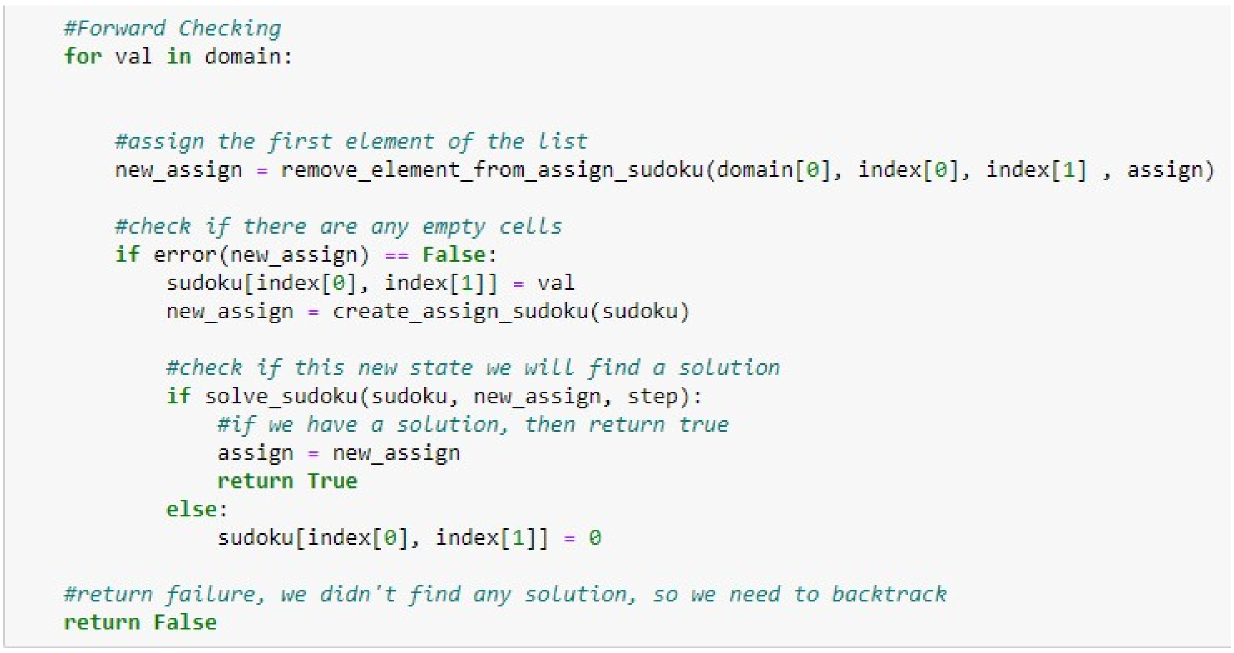
Third Step: we use a technique called the "*Forward Checking*" in order to achieve the results that we need. To do so we iterate for each value on the domain to find a solution trying to assign that value to our sudoku, this will create a *new_assign_matrix*.

One problem that may occur is that assigning a value an array-domain in our *assign_matrix* could be set equal to [ ], caused by removing all occurrence of that value in the domains of the other variables. If that happens we can't add that value to our sudoku or we will not find a solution. So the iteration moves to the next value of the domain, skipping the current value. To do this check it's used a function called "*error*".

Fourth Step: if our *assign_matrix* hasn't any cell = [ ] then we can solve the backtracking call "*solve_sudoku*" on the *new_assign_matrix* and the *new sudoku*. If the result it's true, this mean we found a solution and we can assign the new value to our sudoku. Instead if it returns false, this mean we need to go to the next value of the iterate domain.

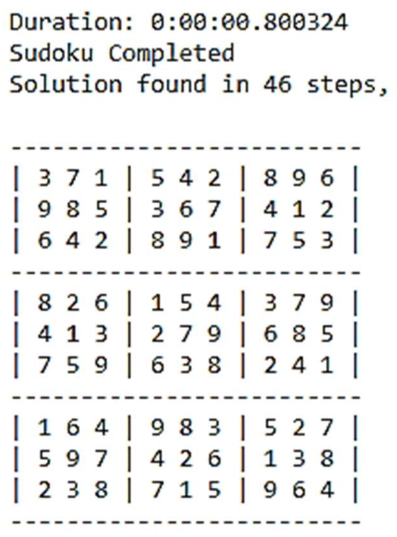
The algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable trying to find a solution. If an inconsistency is detected, then backtrack returns failure, causing the previous call to try another value.

Fifth Step: if our iteration of the domain ends, this mean we didn't find any solution so we return False.



```
    #Forward Checking
    for val in domain:


        #assign the first element of the list
        new_assign = remove_element_from_assign_sudoku(domain[0], index[0], index[1] , assign)

        #check if there are any empty cells
        if error(new_assign) == False:
            sudoku[index[0], index[1]] = val
            new_assign = create_assign_sudoku(sudoku)

            #check if this new state we will find a solution
            if solve_sudoku(sudoku, new_assign, step):
                #if we have a solution, then return true
                assign = new_assign
                return True
            else:
                sudoku[index[0], index[1]] = 0

    #return failure, we didn't find any solution, so we need to backtrack
    return False
```

So in the end this is the solution we found, in 0.8 seconds and 46 steps



```
Duration: 0:00:00.800324
Sudoku Completed
Solution found in 46 steps,

-------------------------
| 3 7 1 | 5 4 2 | 8 9 6 |
| 9 8 5 | 3 6 7 | 4 1 2 |
| 6 4 2 | 8 9 1 | 7 5 3 |
-------------------------
| 8 2 6 | 1 5 4 | 3 7 9 |
| 4 1 3 | 2 7 9 | 6 8 5 |
| 7 5 9 | 6 3 8 | 2 4 1 |
-------------------------
| 1 6 4 | 9 8 3 | 5 2 7 |
| 5 9 7 | 4 2 6 | 1 3 8 |
| 2 3 8 | 7 1 5 | 9 6 4 |
-------------------------
```

Test:

In this section we will see some tests, for each test we the number of steps and the time to find solution.

Test 1:

```
-------------------------
|   1 6 | 3   8 | 4 2   |
| 8 4   |     7 | 3     |
| 3     |       |       |
-------------------------
|   6   | 9 4   | 8   2 |
|   8 1 |   3   | 7 9   |
| 9   3 |   7 6 |   4   |
-------------------------
|       |       |     3 |
|     5 | 7     |   6 8 |
|   7 8 | 1   3 | 2 5   |
-------------------------
Duration: 0:00:00.611177
Sudoku Completed
Solution found in 44 steps,

-------------------------
| 7 1 6 | 3 5 8 | 4 2 9 |
| 8 4 9 | 2 6 7 | 3 1 5 |
| 3 5 2 | 4 1 9 | 6 8 7 |
-------------------------
| 5 6 7 | 9 4 1 | 8 3 2 |
| 4 8 1 | 5 3 2 | 7 9 6 |
| 9 2 3 | 8 7 6 | 5 4 1 |
-------------------------
| 2 9 4 | 6 8 5 | 1 7 3 |
| 1 3 5 | 7 2 4 | 9 6 8 |
| 6 7 8 | 1 9 3 | 2 5 4 |
-------------------------
```

Test 2:

```
-------------------------
|     2 |   3   |     8 |
|       |     8 |       |
|   3 1 |   2   |       |
-------------------------
|   6   |   5   | 2 7   |
|   1   |       |   5   |
| 2   4 |   6   |   3 1 |
-------------------------
|       |   8   | 6   5 |
|       |       |   1 3 |
|     5 | 3 1   | 4     |
-------------------------
Duration: 0:00:02.611870
Sudoku Completed
Solution found in 137 steps,

-------------------------
| 6 7 2 | 4 3 5 | 1 9 8 |
| 5 4 9 | 1 7 8 | 3 6 2 |
| 8 3 1 | 6 2 9 | 5 4 7 |
-------------------------
| 3 6 8 | 9 5 1 | 2 7 4 |
| 9 1 7 | 2 4 3 | 8 5 6 |
| 2 5 4 | 8 6 7 | 9 3 1 |
-------------------------
| 1 9 3 | 7 8 4 | 6 2 5 |
| 4 8 6 | 5 9 2 | 7 1 3 |
| 7 2 5 | 3 1 6 | 4 8 9 |
-------------------------
```

Test 3:

```
-------------------------
|       |       |       |
|     8 |       |   4   |
|       |       |       |
-------------------------
|       |     6 |       |
|       |       |       |
|       |       |       |
-------------------------
| 2     |       |       |
|       |       | 2     |
|       |       |       |
-------------------------
Duration: 0:00:02.044335
Sudoku Completed
Solution found in 76 steps,

-------------------------
| 6 4 5 | 7 8 9 | 1 2 3 |
| 1 2 8 | 5 6 3 | 7 4 9 |
| 3 7 9 | 1 2 4 | 5 6 8 |
-------------------------
| 8 1 2 | 4 7 6 | 3 9 5 |
| 5 6 7 | 3 9 8 | 4 1 2 |
| 9 3 4 | 2 1 5 | 6 8 7 |
-------------------------
| 2 5 6 | 9 3 1 | 8 7 4 |
| 4 9 1 | 8 5 7 | 2 3 6 |
| 7 8 3 | 6 4 2 | 9 5 1 |
-------------------------
```

Test 4:

```
-------------------------
|       |       |       |
|       |       |       |
|       |       |       |
-------------------------
|       |       |       |
|       |       |       |
|       |       |       |
-------------------------
|       |       |       |
|       |       |       |
|       |       |       |
-------------------------
Duration: 0:00:02.183485
Sudoku Completed
Solution found in 81 steps,

-------------------------
| 1 2 3 | 4 5 6 | 7 8 9 |
| 4 5 6 | 7 8 9 | 1 2 3 |
| 7 8 9 | 1 2 3 | 4 5 6 |
-------------------------
| 2 3 1 | 6 7 4 | 8 9 5 |
| 8 7 5 | 9 1 2 | 3 6 4 |
| 6 9 4 | 5 3 8 | 2 1 7 |
-------------------------
| 3 1 7 | 2 6 5 | 9 4 8 |
| 5 4 2 | 8 9 7 | 6 3 1 |
| 9 6 8 | 3 4 1 | 5 7 2 |
-------------------------
```

Test 5:

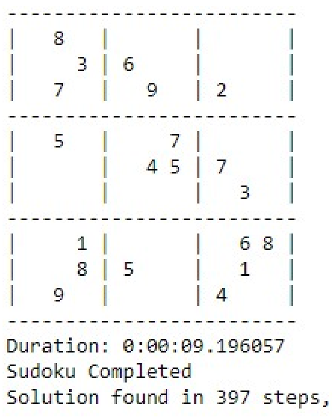
```
-------------------------
|   2   |       |       |
|       | 6     |     3 |
|   7 4 |   8   |       |
-------------------------
|       |     3 |     2 |
|   8   |   4   |   1   |
| 6     | 5     |       |
-------------------------
|       |   1   | 7 8   |
| 5     |       |       |
|       |     9 |   4   |
-------------------------
Duration: 0:01:23.853386
Sudoku Completed
Solution found in 3183 steps,

-------------------------
| 8 2 6 | 1 3 7 | 9 5 4 |
| 1 9 5 | 6 2 4 | 8 7 3 |
| 3 7 4 | 9 8 5 | 1 2 6 |
-------------------------
| 7 5 1 | 8 9 3 | 4 6 2 |
| 9 8 3 | 2 4 6 | 5 1 7 |
| 6 4 2 | 5 7 1 | 3 9 8 |
-------------------------
| 4 6 9 | 3 1 2 | 7 8 5 |
| 5 1 7 | 4 6 8 | 2 3 9 |
| 2 3 8 | 7 5 9 | 6 4 1 |
-------------------------
```

Test 6:

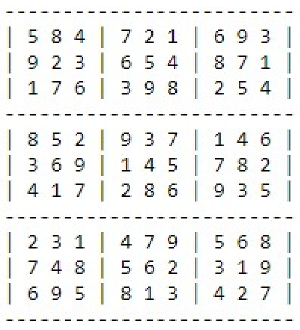
```
-------------------------
| 8     |       |       |
|     3 | 6     |       |
|   7   |   9   | 2     |
-------------------------
|   5   |     7 |       |
|       |   4 5 | 7     |
|       | 1     |   3   |
-------------------------
|     1 |       |   6 8 |
|     8 | 5     |   1   |
|   9   |       | 4     |
-------------------------
Duration: 0:03:34.200702
Sudoku Completed
Solution found in 9179 steps,

-------------------------
| 8 1 2 | 7 5 3 | 6 4 9 |
| 9 4 3 | 6 8 2 | 1 7 5 |
| 6 7 5 | 4 9 1 | 2 8 3 |
-------------------------
| 1 5 4 | 2 3 7 | 8 9 6 |
| 3 6 9 | 8 4 5 | 7 2 1 |
| 2 8 7 | 1 6 9 | 5 3 4 |
-------------------------
| 5 2 1 | 9 7 4 | 3 6 8 |
| 4 3 8 | 5 2 6 | 9 1 7 |
| 7 9 6 | 3 1 8 | 4 5 2 |
-------------------------
```

Test 5 and Test 6 need much more time to be solved then the other sudokus. This is caused by a problem called overhead. Backtracking is very expensive and we do it on every domain value of every variable. As we have said, back tracking performs a depth-first search to find the solution, but if it is located in the rightmost node it will only overhead the search times. We notice this aspect thanks to the number of steps shown in the two images of test 5 and test 6, the previous ones require less than 100 steps instead with the last two tests we go from 3000 steps for test 5 to 9000 steps for test 6.

The last test that we try will be the “Hardest-Ever Sudoku”, which has only one solution:



```
-------------------------
|   8   |       |       |
|     3 | 6     |       |
|   7   |   9   | 2     |
-------------------------
|   5   |     7 |       |
|       |   4 5 | 7     |
|       |       |   3   |
-------------------------
|     1 |       |   6 8 |
|     8 | 5     |   1   |
|   9   |       | 4     |
-------------------------
Duration: 0:00:09.196057
Sudoku Completed
Solution found in 397 steps,
```



```
-------------------------
| 5 8 4 | 7 2 1 | 6 9 3 |
| 9 2 3 | 6 5 4 | 8 7 1 |
| 1 7 6 | 3 9 8 | 2 5 4 |
-------------------------
| 8 5 2 | 9 3 7 | 1 4 6 |
| 3 6 9 | 1 4 5 | 7 8 2 |
| 4 1 7 | 2 8 6 | 9 3 5 |
-------------------------
| 2 3 1 | 4 7 9 | 5 6 8 |
| 7 4 8 | 5 6 2 | 3 1 9 |
| 6 9 5 | 8 1 3 | 4 2 7 |
-------------------------
```

Conclusion:

In this paper we have discussed a constraint formulation to solve the Sudoku puzzle, we used *Minimum-remaining-values (MRV) heuristic* and *Forward Checking* as a criteria of selection of variables and values, thanks to them we avoided to do unnecessary calculations that force to backtracking

We have seen that most of the time we can solve a sudoku in seconds. The only problem we encountered was in some tests where the time of solving sudoku excides 2 or 3 minutes. This is caused by the *Overhead Problem*. The search time increased due to the analysis used in *Forward Checking* that we applied to solve the problem.

Simulated Annealing

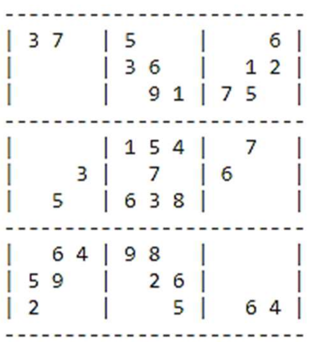
Definition:

Simulated annealing (SA) is a stochastic approach that simulates the statistical process to reach its global minimum. *The basic procedure for implementation of this analogy to the annealing process* is to generate random points in the neighborhood and evaluate the cost functions there. If the cost function value is smaller than its current best value, then the point is accepted, and the best function value is updated. If the function value is higher than the best value known thus far, then the point is sometimes accepted and sometimes rejected. Point's acceptance is based on the value of the probability density function. If this probability density function has a value greater than a random number, then the trial point is accepted as the best solution even if its function value is higher than the known best value. In computing the probability density function, a parameter called the *temperature* is used.

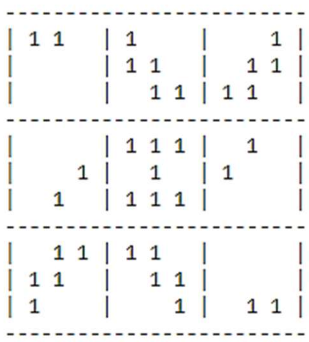
Initially, a larger temperature is selected. As the trials progress, the temperature is reduced (this is called the *cooling schedule*). The acceptance probability steadily decreases to zero as the temperature is reduced. Thus, in the initial stages, the method sometimes accepts worse designs, while in the final stages, the worse designs are almost always rejected. This strategy avoids getting trapped at a local minimum point.

Sudoku:

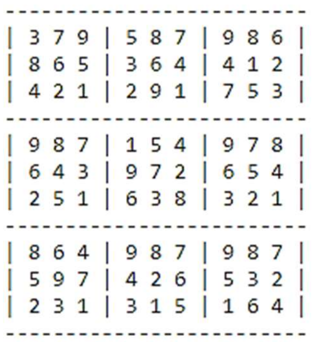
In order to form an initial candidate solution, we fill the grid by assigning each non-fixed cell (cells of sudoku that can be fill) in the grid a value. This is done randomly, but in such a way so that when the grid is full, every square contains the values 1 to 9 exactly once. The following images show the sudoku to be solved, the *fixed_value_matrix* and the *random_fill_matrix*.



```
-------------------------
| 3 7   | 5     |     6 |
|       | 3 6   |   1 2 |
|       |   9 1 | 7 5   |
-------------------------
|       | 1 5 4 |   7   |
|     3 |   7   | 6     |
|   5   | 6 3 8 |       |
-------------------------
|   6 4 | 9 8   |       |
| 5 9   |   2 6 |       |
| 2     |     5 |   6 4 |
-------------------------
```



```
-------------------------
| 1 1   | 1     |     1 |
|       | 1 1   |   1 1 |
|       |   1 1 | 1 1   |
-------------------------
|       | 1 1 1 |   1   |
|     1 |   1   | 1     |
|   1   | 1 1 1 |       |
-------------------------
|   1 1 | 1 1   |       |
| 1 1   |   1 1 |       |
| 1     |     1 |   1 1 |
-------------------------
```

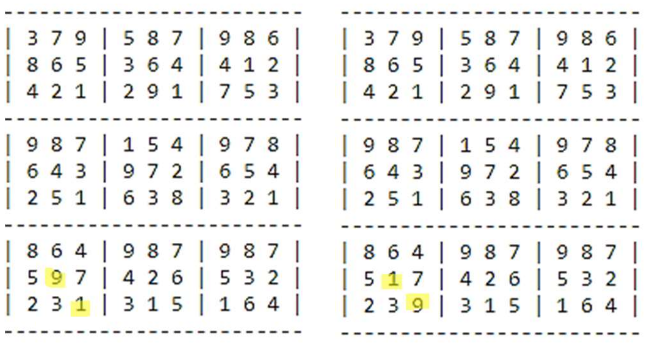


```
-------------------------
| 3 7 9 | 5 8 7 | 9 8 6 |
| 8 6 5 | 3 6 4 | 4 1 2 |
| 4 2 1 | 2 9 1 | 7 5 3 |
-------------------------
| 9 8 7 | 1 5 4 | 9 7 8 |
| 6 4 3 | 9 7 2 | 6 5 4 |
| 2 5 1 | 6 3 8 | 3 2 1 |
-------------------------
| 8 6 4 | 9 8 7 | 9 8 7 |
| 5 9 7 | 4 2 6 | 5 3 2 |
| 2 3 1 | 3 1 5 | 1 6 4 |
-------------------------
```

During the run, the neighborhood operator repeatedly chooses two different non-fixed cells in the same square and swaps them.

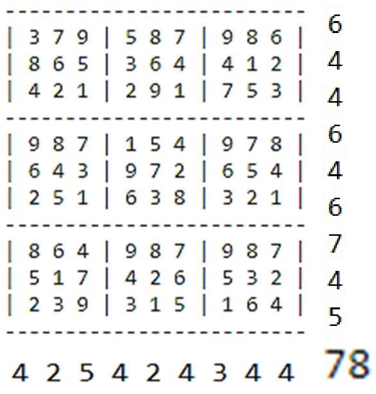
1. Choose randomly i and j, such that 1 ≤ i, j ≤ 9 , and cell(i, j) is non-fixed.
2. Choose randomly k and l, such that (a) cell(k, l) is in the same square as cell(i, j), cell(k, l) is non-fixed, and cell(k,l) ≠ cell(i, j).
3. Swap cell(i, j) with cell(k, l).

This method of producing an initial candidate solution, together with the defined neighborhood operator, ensures that the third criterion of the puzzle is always met. As we see in the following images we swap cell(7, 2) with cell(8, 3).



```
-------------------------    -------------------------
| 3 7 9 | 5 8 7 | 9 8 6 |    | 3 7 9 | 5 8 7 | 9 8 6 |
| 8 6 5 | 3 6 4 | 4 1 2 |    | 8 6 5 | 3 6 4 | 4 1 2 |
| 4 2 1 | 2 9 1 | 7 5 3 |    | 4 2 1 | 2 9 1 | 7 5 3 |
-------------------------    -------------------------
| 9 8 7 | 1 5 4 | 9 7 8 |    | 9 8 7 | 1 5 4 | 9 7 8 |
| 6 4 3 | 9 7 2 | 6 5 4 |    | 6 4 3 | 9 7 2 | 6 5 4 |
| 2 5 1 | 6 3 8 | 3 2 1 |    | 2 5 1 | 6 3 8 | 3 2 1 |
-------------------------    -------------------------
| 8 6 4 | 9 8 7 | 9 8 7 |    | 8 6 4 | 9 8 7 | 9 8 7 |
| 5 9 7 | 4 2 6 | 5 3 2 |    | 5 1 7 | 4 2 6 | 5 3 2 |
| 2 3 1 | 3 1 5 | 1 6 4 |    | 2 3 9 | 3 1 5 | 1 6 4 |
-------------------------    -------------------------
```

Because our representation and neighborhood operator ensures that the third criterion of sudoku is always met (see sudoku definition), an appropriate *cost function* is obviously one that inspects for violations of the remaining two criteria (see sudoku definition). The we use our *cost function* to look at each row individually and calculates the number of values, 1 through to 9 that are not present. The same is then done for each column, and the cost is simply the total of these values. Obviously, an optimal solution will have a cost of zero.



```
-------------------------   6
| 3 7 9 | 5 8 7 | 9 8 6 |   4
| 8 6 5 | 3 6 4 | 4 1 2 |   4
| 4 2 1 | 2 9 1 | 7 5 3 |
-------------------------   6
| 9 8 7 | 1 5 4 | 9 7 8 |   4
| 6 4 3 | 9 7 2 | 6 5 4 |   6
| 2 5 1 | 6 3 8 | 3 2 1 |
-------------------------   7
| 8 6 4 | 9 8 7 | 9 8 7 |   4
| 5 1 7 | 4 2 6 | 5 3 2 |   5
| 2 3 9 | 3 1 5 | 1 6 4 |
-------------------------
 4 2 5 4 2 4 3 4 4          78
```

In this approach a single application of the above neighborhood operator means that at most two rows of cells and two columns of cells are affected by a swap. Thus, after a single move in the neighborhood, rather than re-evaluate the whole candidate solution, only the cells that have "*row_cost*" or "*column_cost*" equal to 0 are swapped. Such a scheme offers a considerable speed-up to the algorithm.

We need also to choose the initial temperature T_0, that should allow the majority of moves to be accepted and then, during the run, it should be slowly reduced (according to a *cooling schedule*) so that the algorithm becomes increasingly greedy. If the algorithm is successful, eventually the search should converge at (or near to) a *global optimum*. However, T_0 should be chosen with care: a value that is too high will mean that computation time is wasted because the algorithm is likely to conduct an unhelpful random walk through the search space. On the other hand, a value for T_0 that is too low will also have a negative impact, as it will make the search too greedy from the outset, and therefore make it more susceptible to getting stuck in *local minimum*.

During the run, the temperature is reduced using a simple *geometric cooling* schedule where the current temperature T_i is modified to a new temperature T_{i+1} via the formula:

$$T_{i+1} + = \alpha \cdot T_i$$

where a is a control parameter known as the *cooling rate*, and 0 < a < 1. Obviously, a large value for a, such as 0.999, will cause the temperature to drop very slowly, whilst a setting of, say, 0.5 will cause a much quicker cooling.

Having defined a suitable representation, neighborhood operator, and evaluation function, the application of the *Simulated Annealing* (SA) metaheuristic is now straightforward.

First Step: create *FixedSudoku* fixing all the values of the sudoku to 1 if it's present a number not equal to 0. Next we calculated the number of iterations, which is calculated by counting how many times 1 occur in the FixedSudoku. The variable *solutionFound* stops the while loop when the solution is found.

Second Step: choose the initial Temperature T_0, after some test the best parameter that I found is T = 0.4 and for the *decreaseFactor* (*cooling factor*) = 0.999, so the temperature will decrease slowly.

Third Step: random fill the sudoku and calculate the *Cost Function* on it. As we see the *Cost Function* is the score of the puzzle. The cost of each number is dictated by the quantity of the same number in the same row and column.

```
def solveSudoku (sudoku):
    #criteria to stop the lopp
    solutionFound = 0

    #fix the value of sudoku to 1
    fixedSudoku = np.copy(sudoku)
    FixValues(fixedSudoku)

    #number of value of fixedSudoku = 1
    itterations = ChooseNumberOfItterations(fixedSudoku)

    #steps count
    step = 0

    # = 0 solution not found, = 1 solution found
    while (solutionFound == 0):

        #cooling factor
        decreaseFactor = 0.999


        stuckCount = 0

        #random fill the matrix
        tmpSudoku = random_fill_matrix(np.copy(sudoku))
        score = score_board(tmpSudoku)

        #temperature
        T = 0.4

        #Cost Function Check
        if score <= 0:
            solutionFound = 1
```

Fourth Step: Starting with an initial candidate solution, an exploration of the search space is conducted by iteratively applying the above neighborhood operator. Calculate the score of the new state, and then the old score is compared to the new score. Two reason to choose a new state:

1. The old score is greater than the new score *new score < old score*.
2. If the new score is higher than the old, we can still keep it, to find out we use the *Boltzmann Distribution*

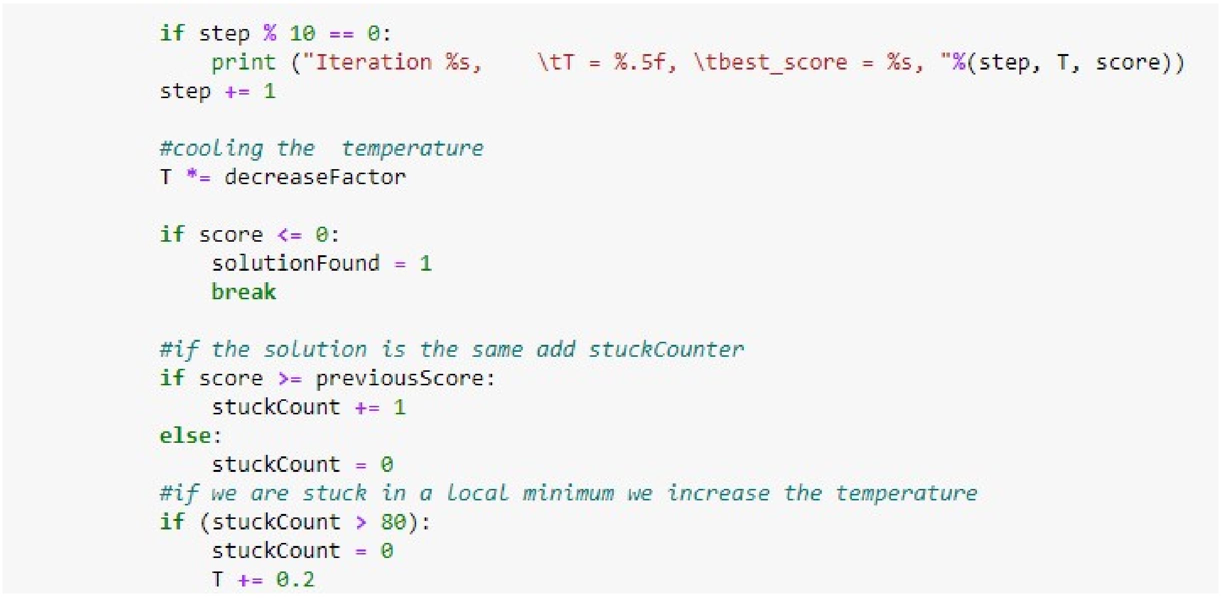
   "exp(-(new score – old score)/ T)"

   the difference is the energy for the system divide by T the temperature and we compare with a random number. If the random number is smaller than this value we keep the new state.

```
        #search for a new state
        for i in range (0, itterations+1):

            tmpSudoku = ChooseNewState(tmpSudoku, fixedSudoku, T)

            score = score_board(tmpSudoku)
            if score <= 0:
                solutionFound = 1
                break
```

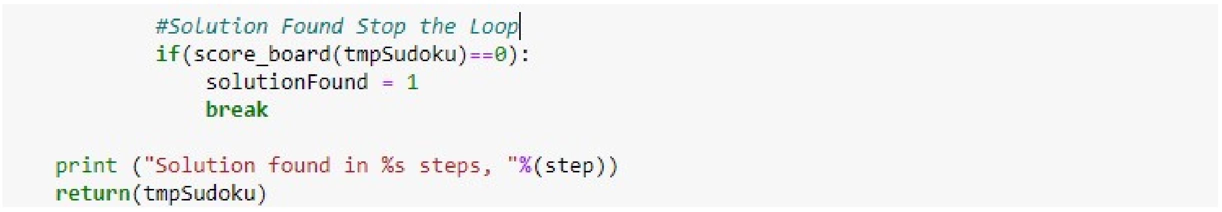
```
def ChooseNewState (currentSudoku, fixedSudoku, sigma):
    newSudoku = random_swap_two_element(currentSudoku, fixedSudoku)
    currentCost = score_board(currentSudoku)
    newCost = score_board(newSudoku)
    costDifference = newCost - currentCost
    rho = math.exp(-costDifference/sigma)
    if(np.random.uniform(1,0,1) < rho):
        return newSudoku
    return currentSudoku
```

Fifth Step: We also have a *stuckCount* because our solver could be stuck in a *local minimum* and it will never change state. To solve this problem we count how much the same value repeat and if this value exceeds 80 we increase the temperature by 0.2 so we can move from the current state and try to reach the global minimum. The modification investigated allows the simplex to escape from local minima, and also to avoid deviation and repeated stalls into non-optimal regions.



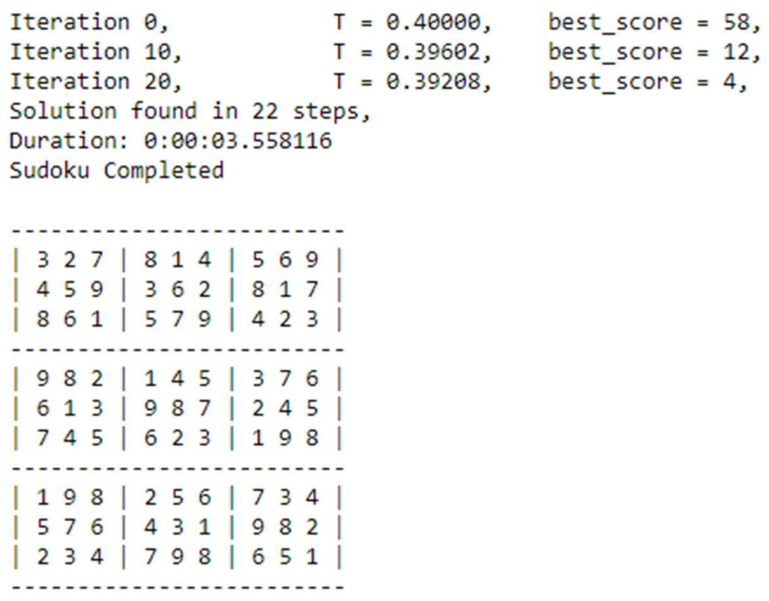
```
        if step % 10 == 0:
            print ("Iteration %s,    \tT = %.5f, \tbest_score = %s, "%(step, T, score))
        step += 1

        #cooling the  temperature
        T *= decreaseFactor

        if score <= 0:
            solutionFound = 1
            break

        #if the solution is the same add stuckCounter
        if score >= previousScore:
            stuckCount += 1
        else:
            stuckCount = 0
        #if we are stuck in a local minimum we increase the temperature
        if (stuckCount > 80):
            stuckCount = 0
            T += 0.2
```

Sixth Step: At the end if we reach our termination criteria, the cost of the state is equal to 0, this mean we reach the global minimum and set solutionFound = 1 so we stop the loop and can give in output our solution.



```
        #Solution Found Stop the Loop
        if(score_board(tmpSudoku)==0):
            solutionFound = 1
            break

    print ("Solution found in %s steps, "%(step))
    return(tmpSudoku)
```

So in the end in 3.5 second and 22 steps we found the solution on this problem:

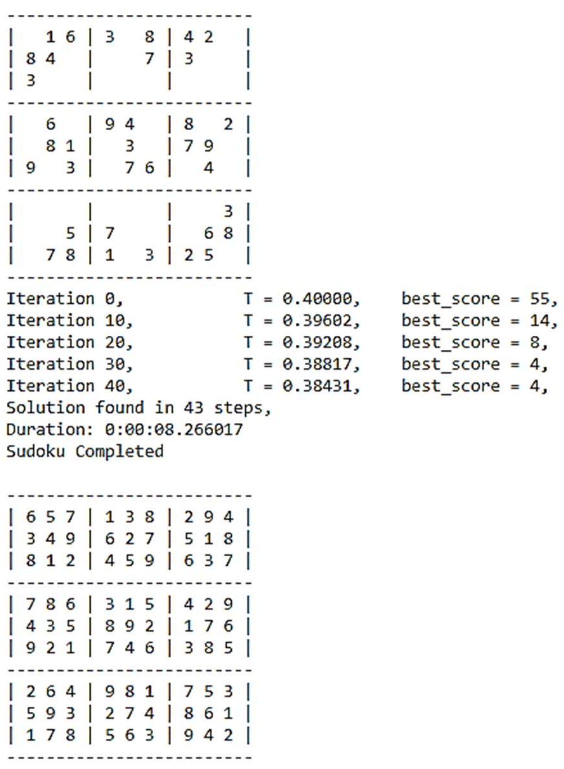


```
Iteration 0,            T = 0.40000,    best_score = 58,
Iteration 10,           T = 0.39602,    best_score = 12,
Iteration 20,           T = 0.39208,    best_score = 4,
Solution found in 22 steps,
Duration: 0:00:03.558116
Sudoku Completed

-------------------------
| 3 2 7 | 8 1 4 | 5 6 9 |
| 4 5 9 | 3 6 2 | 8 1 7 |
| 8 6 1 | 5 7 9 | 4 2 3 |
-------------------------
| 9 8 2 | 1 4 5 | 3 7 6 |
| 6 1 3 | 9 8 7 | 2 4 5 |
| 7 4 5 | 6 2 3 | 1 9 8 |
-------------------------
| 1 9 8 | 2 5 6 | 7 3 4 |
| 5 7 6 | 4 3 1 | 9 8 2 |
| 2 3 4 | 7 9 8 | 6 5 1 |
-------------------------
```

Test:

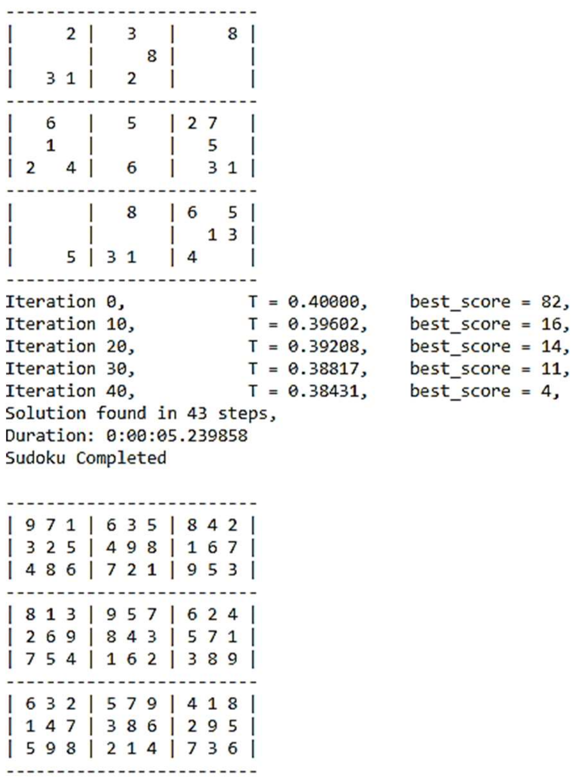
In this section we will see some tests, for each test we the number of steps and the time to find solution.

Test 1:



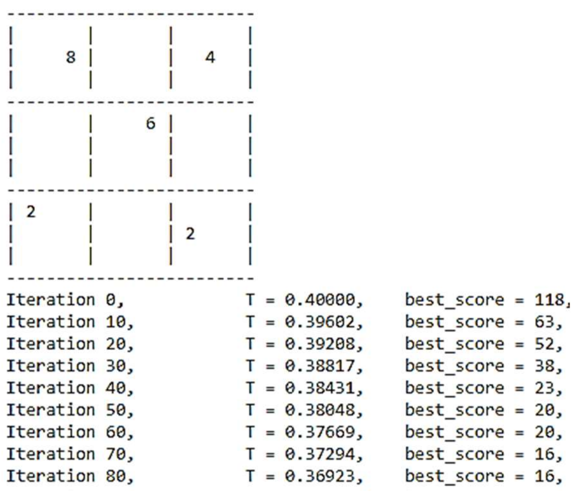
```
-------------------------
|   1 6 | 3   8 | 4 2   |
| 8 4   |     7 | 3     |
| 3     |       |       |
-------------------------
|   6   | 9 4   | 8   2 |
|   8 1 |   3   | 7 9   |
| 9   3 |   7 6 |   4   |
-------------------------
|       |       |     3 |
|     5 | 7     |   6 8 |
|   7 8 | 1   3 | 2 5   |
-------------------------
Iteration 0,            T = 0.40000,    best_score = 55,
Iteration 10,           T = 0.39602,    best_score = 14,
Iteration 20,           T = 0.39208,    best_score = 8,
Iteration 30,           T = 0.38817,    best_score = 4,
Iteration 40,           T = 0.38431,    best_score = 4,
Solution found in 43 steps,
Duration: 0:00:08.266017
Sudoku Completed

-------------------------
| 6 5 7 | 1 3 8 | 2 9 4 |
| 3 4 9 | 6 2 7 | 5 1 8 |
| 8 1 2 | 4 5 9 | 6 3 7 |
-------------------------
| 7 8 6 | 3 1 5 | 4 2 9 |
| 4 3 5 | 8 9 2 | 1 7 6 |
| 9 2 1 | 7 4 6 | 3 8 5 |
-------------------------
| 2 6 4 | 9 8 1 | 7 5 3 |
| 5 9 3 | 2 7 4 | 8 6 1 |
| 1 7 8 | 5 6 3 | 9 4 2 |
-------------------------
```

Test 2:

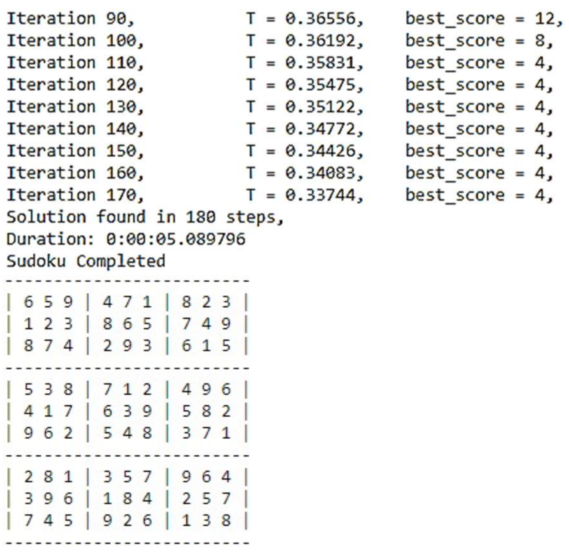


```
-------------------------
|     2 |   3   |     8 |
|       |     8 |       |
|   3 1 |   2   |       |
-------------------------
|   6   |   5   | 2 7   |
|   1   |       |   5   |
| 2   4 |   6   |   3 1 |
-------------------------
|       |   8   | 6   5 |
|       |       |   1 3 |
|     5 | 3 1   | 4     |
-------------------------
Iteration 0,            T = 0.40000,    best_score = 82,
Iteration 10,           T = 0.39602,    best_score = 16,
Iteration 20,           T = 0.39208,    best_score = 14,
Iteration 30,           T = 0.38817,    best_score = 11,
Iteration 40,           T = 0.38431,    best_score = 4,
Solution found in 43 steps,
Duration: 0:00:05.239858
Sudoku Completed

-------------------------
| 9 7 1 | 6 3 5 | 8 4 2 |
| 3 2 5 | 4 9 8 | 1 6 7 |
| 4 8 6 | 7 2 1 | 9 5 3 |
-------------------------
| 8 1 3 | 9 5 7 | 6 2 4 |
| 2 6 9 | 8 4 3 | 5 7 1 |
| 7 5 4 | 1 6 2 | 3 8 9 |
-------------------------
| 6 3 2 | 5 7 9 | 4 1 8 |
| 1 4 7 | 3 8 6 | 2 9 5 |
| 5 9 8 | 2 1 4 | 7 3 6 |
-------------------------
```

Test 3:

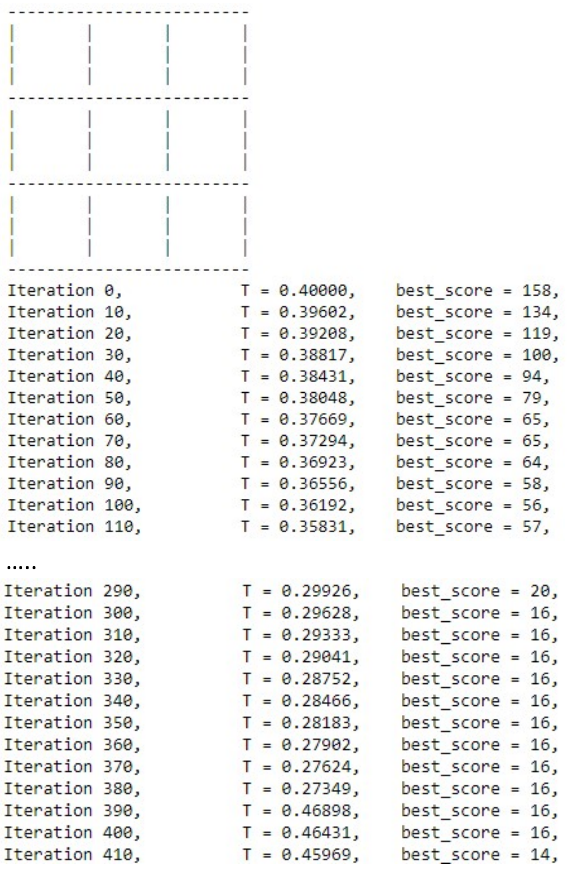


```
-------------------------
|       |       |       |
|     8 |       |   4   |
|       |       |       |
-------------------------
|       |     6 |       |
|       |       |       |
|       |       |       |
-------------------------
| 2     |       |       |
|       |       | 2     |
|       |       |       |
-------------------------
Iteration 0,            T = 0.40000,    best_score = 118,
Iteration 10,           T = 0.39602,    best_score = 63,
Iteration 20,           T = 0.39208,    best_score = 52,
Iteration 30,           T = 0.38817,    best_score = 38,
Iteration 40,           T = 0.38431,    best_score = 23,
Iteration 50,           T = 0.38048,    best_score = 20,
Iteration 60,           T = 0.37669,    best_score = 20,
Iteration 70,           T = 0.37294,    best_score = 16,
Iteration 80,           T = 0.36923,    best_score = 16,
```



```
Iteration 90,           T = 0.36556,    best_score = 12,
Iteration 100,          T = 0.36192,    best_score = 8,
Iteration 110,          T = 0.35831,    best_score = 4,
Iteration 120,          T = 0.35475,    best_score = 4,
Iteration 130,          T = 0.35122,    best_score = 4,
Iteration 140,          T = 0.34772,    best_score = 4,
Iteration 150,          T = 0.34426,    best_score = 4,
Iteration 160,          T = 0.34083,    best_score = 4,
Iteration 170,          T = 0.33744,    best_score = 4,
Solution found in 180 steps,
Duration: 0:00:05.089796
Sudoku Completed
-------------------------
| 6 5 9 | 4 7 1 | 8 2 3 |
| 1 2 3 | 8 6 5 | 7 4 9 |
| 8 7 4 | 2 9 3 | 6 1 5 |
-------------------------
| 5 3 8 | 7 1 2 | 4 9 6 |
| 4 1 7 | 6 3 9 | 5 8 2 |
| 9 6 2 | 5 4 8 | 3 7 1 |
-------------------------
| 2 8 1 | 3 5 7 | 9 6 4 |
| 3 9 6 | 1 8 4 | 2 5 7 |
| 7 4 5 | 9 2 6 | 1 3 8 |
-------------------------
```

Test 4:

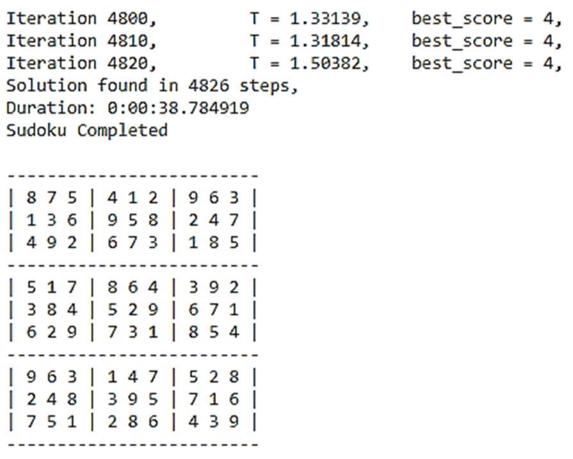


```
-------------------------
|       |       |       |
|       |       |       |
|       |       |       |
-------------------------
|       |       |       |
|       |       |       |
|       |       |       |
-------------------------
|       |       |       |
|       |       |       |
|       |       |       |
-------------------------
Iteration 0,            T = 0.40000,    best_score = 158,
Iteration 10,           T = 0.39602,    best_score = 134,
Iteration 20,           T = 0.39208,    best_score = 119,
Iteration 30,           T = 0.38817,    best_score = 100,
Iteration 40,           T = 0.38431,    best_score = 94,
Iteration 50,           T = 0.38048,    best_score = 79,
Iteration 60,           T = 0.37669,    best_score = 65,
Iteration 70,           T = 0.37294,    best_score = 65,
Iteration 80,           T = 0.36923,    best_score = 64,
Iteration 90,           T = 0.36556,    best_score = 58,
Iteration 100,          T = 0.36192,    best_score = 56,
Iteration 110,          T = 0.35831,    best_score = 57,
```

.....

```
Iteration 290,          T = 0.29926,    best_score = 20,
Iteration 300,          T = 0.29628,    best_score = 16,
Iteration 310,          T = 0.29333,    best_score = 16,
Iteration 320,          T = 0.29041,    best_score = 16,
Iteration 330,          T = 0.28752,    best_score = 16,
Iteration 340,          T = 0.28466,    best_score = 16,
Iteration 350,          T = 0.28183,    best_score = 16,
Iteration 360,          T = 0.27902,    best_score = 16,
Iteration 370,          T = 0.27624,    best_score = 16,
Iteration 380,          T = 0.27349,    best_score = 16,
Iteration 390,          T = 0.46898,    best_score = 16,
Iteration 400,          T = 0.46431,    best_score = 16,
Iteration 410,          T = 0.45969,    best_score = 14,
```

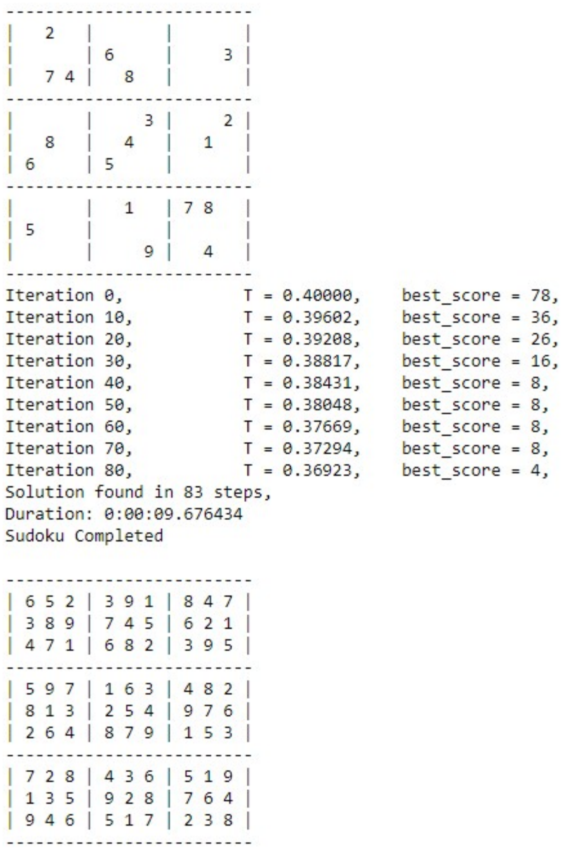
....



```
Iteration 4800,         T = 1.33139,    best_score = 4,
Iteration 4810,         T = 1.31814,    best_score = 4,
Iteration 4820,         T = 1.50382,    best_score = 4,
Solution found in 4826 steps,
Duration: 0:00:38.784919
Sudoku Completed

-------------------------
| 8 7 5 | 4 1 2 | 9 6 3 |
| 1 3 6 | 9 5 8 | 2 4 7 |
| 4 9 2 | 6 7 3 | 1 8 5 |
-------------------------
| 5 1 7 | 8 6 4 | 3 9 2 |
| 3 8 4 | 5 2 9 | 6 7 1 |
| 6 2 9 | 7 3 1 | 8 5 4 |
-------------------------
| 9 6 3 | 1 4 7 | 5 2 8 |
| 2 4 8 | 3 9 5 | 7 1 6 |
| 7 5 1 | 2 8 6 | 4 3 9 |
-------------------------
```

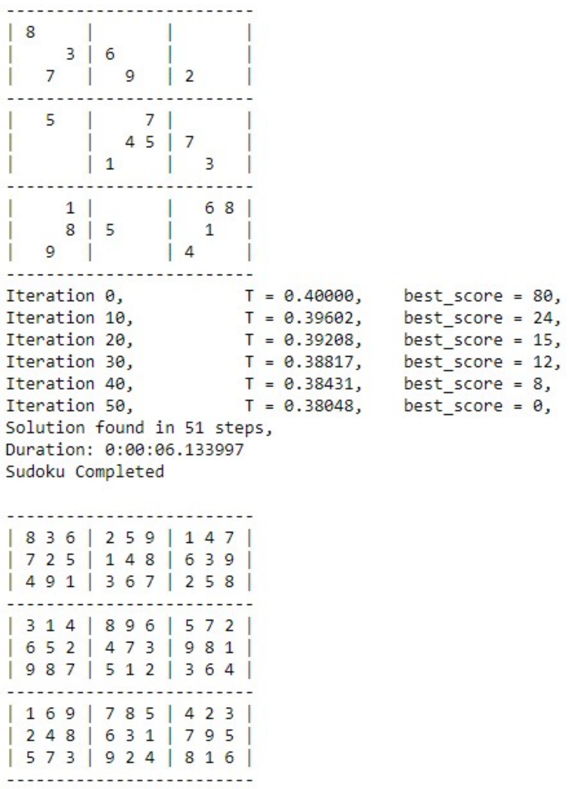
From interaction 300 to iteration 380 we always have the same score, this can only mean that the algorithm is stuck in a local minimum. To sidestep this problem, we need to raise our temperature, so the algorithm chooses a new state. Previously we saw how the variable "StuckCount" in case there were 80 same repetitions would increase the current temperature by 0.2, in fact in the image we see the temperature pass from T = 0.27349 to T = 0.46898..

Test 5:



```
-------------------------
|   2   |       |       |
|       | 6     |     3 |
|   7 4 |   8   |       |
-------------------------
|       |     3 |     2 |
|   8   |   4   |   1   |
| 6     | 5     |       |
-------------------------
|       |   1   | 7 8   |
| 5     |       |       |
|       |     9 |   4   |
-------------------------
Iteration 0,            T = 0.40000,    best_score = 78,
Iteration 10,           T = 0.39602,    best_score = 36,
Iteration 20,           T = 0.39208,    best_score = 26,
Iteration 30,           T = 0.38817,    best_score = 16,
Iteration 40,           T = 0.38431,    best_score = 8,
Iteration 50,           T = 0.38048,    best_score = 8,
Iteration 60,           T = 0.37669,    best_score = 8,
Iteration 70,           T = 0.37294,    best_score = 8,
Iteration 80,           T = 0.36923,    best_score = 4,
Solution found in 83 steps,
Duration: 0:00:09.676434
Sudoku Completed

-------------------------
| 6 5 2 | 3 9 1 | 8 4 7 |
| 3 8 9 | 7 4 5 | 6 2 1 |
| 4 7 1 | 6 8 2 | 3 9 5 |
-------------------------
| 5 9 7 | 1 6 3 | 4 8 2 |
| 8 1 3 | 2 5 4 | 9 7 6 |
| 2 6 4 | 8 7 9 | 1 5 3 |
-------------------------
| 7 2 8 | 4 3 6 | 5 1 9 |
| 1 3 5 | 9 2 8 | 7 6 4 |
| 9 4 6 | 5 1 7 | 2 3 8 |
-------------------------
```

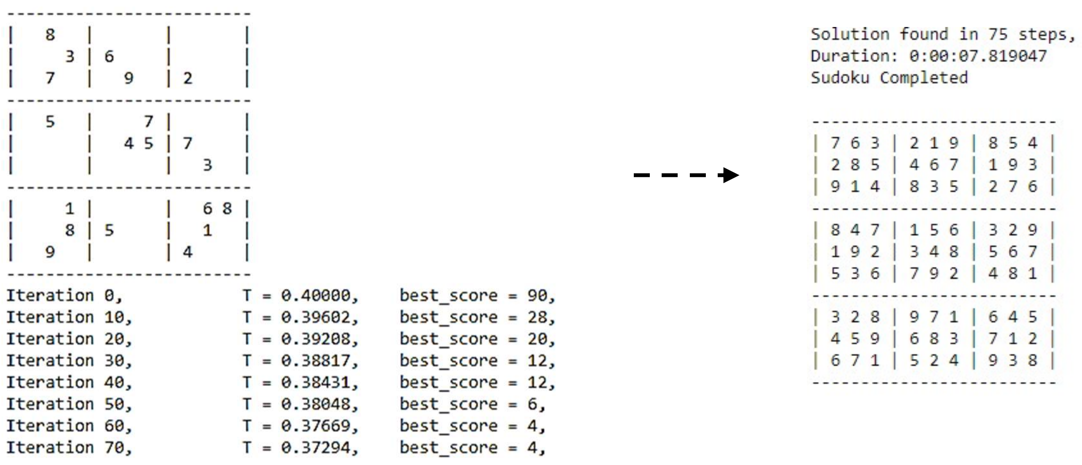
Test 6:



```
-------------------------
| 8     |       |       |
|     3 | 6     |       |
|   7   |   9   | 2     |
-------------------------
|   5   |     7 |       |
|       |   4 5 | 7     |
|       | 1     |   3   |
-------------------------
|     1 |       |   6 8 |
|     8 | 5     |   1   |
|   9   |       | 4     |
-------------------------
Iteration 0,            T = 0.40000,    best_score = 80,
Iteration 10,           T = 0.39602,    best_score = 24,
Iteration 20,           T = 0.39208,    best_score = 15,
Iteration 30,           T = 0.38817,    best_score = 12,
Iteration 40,           T = 0.38431,    best_score = 8,
Iteration 50,           T = 0.38048,    best_score = 0,
Solution found in 51 steps,
Duration: 0:00:06.133997
Sudoku Completed

-------------------------
| 8 3 6 | 2 5 9 | 1 4 7 |
| 7 2 5 | 1 4 8 | 6 3 9 |
| 4 9 1 | 3 6 7 | 2 5 8 |
-------------------------
| 3 1 4 | 8 9 6 | 5 7 2 |
| 6 5 2 | 4 7 3 | 9 8 1 |
| 9 8 7 | 5 1 2 | 3 6 4 |
-------------------------
| 1 6 9 | 7 8 5 | 4 2 3 |
| 2 4 8 | 6 3 1 | 7 9 5 |
| 5 7 3 | 9 2 4 | 8 1 6 |
-------------------------
```

As for test 5 and test 6, unlike the CSP, we don't see any problems in terms of execution. This is because simulated annealing relies on randomness and the only problem is getting stuck in a local minimum.

The last test that we try will be the "Hardest-Ever Sudoku", which has only one solution:



```
-------------------------
|   8   |       |       |
|     3 | 6     |       |
|   7   |   9   | 2     |
-------------------------
|   5   |     7 |       |
|       |   4 5 | 7     |
|       |       |   3   |
-------------------------
|     1 |       |   6 8 |
|     8 | 5     |   1   |
|   9   |       | 4     |
-------------------------
Iteration 0,             T = 0.40000,    best_score = 90,
Iteration 10,            T = 0.39602,    best_score = 28,
Iteration 20,            T = 0.39208,    best_score = 20,
Iteration 30,            T = 0.38817,    best_score = 12,
Iteration 40,            T = 0.38431,    best_score = 12,
Iteration 50,            T = 0.38048,    best_score = 6,
Iteration 60,            T = 0.37669,    best_score = 4,
Iteration 70,            T = 0.37294,    best_score = 4,
```

```
Solution found in 75 steps,
Duration: 0:00:07.819047
Sudoku Completed

-------------------------
| 7 6 3 | 2 1 9 | 8 5 4 |
| 2 8 5 | 4 6 7 | 1 9 3 |
| 9 1 4 | 8 3 5 | 2 7 6 |
-------------------------
| 8 4 7 | 1 5 6 | 3 2 9 |
| 1 9 2 | 3 4 8 | 5 6 7 |
| 5 3 6 | 7 9 2 | 4 8 1 |
-------------------------
| 3 2 8 | 9 7 1 | 6 4 5 |
| 4 5 9 | 6 8 3 | 7 1 2 |
| 6 7 1 | 5 2 4 | 9 3 8 |
-------------------------
```

Conclusion:

We have seen in this paper that this sort of stochastic search approach is able to solve each sudoku, even the hardest. The previous tests (both CSP's and Simulated Annealing's) also shows that in case of sudokus which in CSP – backtracking cause an overhead and the time of computation is very high, with to Simulated Annealing approach we can overcome this problem.

However this approach is completely random and the calculation time can vary according to the temperature and the cooling procedure. Even if we have the same parameters the time may differ due to the randomness of the assignment of values.

This adds another aspect to take into consideration, we are forced to fine-tune the parameters, to do so we need to do more tests, if we choose the wrong hyperparameters the calculation time to solve a sudoku would be exponential.