Università
Ca'Foscari
Venezia

Master's Degree Programme in Computer Science and Information Technology
Software Development and Engineering

1st assignement

Artificial Intelligence: Sudoku Resolver

Student
Andrea Munarin
Matriculation Number 879607

Academic Year
2022-2023

Indice

Capitolo 1

Problem formalization

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

- each number appears only once for each row column and 3×3 box;
- each row, column, and 3×3 box should contain all 9 digits;

The Sudoku puzzle has always some preinserted values. The real challenge for the user is to find all the other numbers that fill the cells respecting all the constraints:

Figura 1.1: Sudoku Example

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

(a) *Example of a sudoku puzzle*

In this first assignment, the purpose is to write a solver for the Sudoku puzzle. The solver should take as input a matrix where empty squares are represented by a standard symbol, in this case 0, while known square should be represented by the corresponding digit (1,. . . ,9).

To do this, we have to use an approach based on constraint propagation and backtracking for the first solution and one of these methods for the second:

1. ***simulated annealing***;
2. ***genetic algorithms***;
3. continuous optimization using ***gradient projection***.

The method choose for this project is the simulated annealing, that will be discussed in the next chapter.

Before starting with a description of the main concepts behind the method for constraint and optimization problems, the first point is to formalized our constraint satisfaction problems. A constraint satisfaction problem (***CSP***) is a set of problems in which the main purpose is to find the solutions (inside a certain domain of values) for each variable we are considering, respecting all the constraints defined.
So a CSP can be formalized in terms of

- **Variables**: represents the entities of our problem. Every entity can be associated to a value present in the Domain
- **Domains**: the set of all the values that can be associated to a variable.
- **Constraints**: the set of the condition that every variable must respect. Define how values can be assigned to variables

The Sudoku puzzle can be seen as a CSP, and in fact we can formalize the problem as follows:

- **Variables**: every cell in a 9×9 board
- **Domains**: every integer number between 0 and 9 includes
- **Constrains**:
  - each number appears only once for each row column and 3×3box;
  - each row, column, and 3×3 box should contain all 9 digits.

Capitolo 2

Theory concepts

2.0.1 Constraint Propagation

To solve CSP problems we need combination of

- **Constraint propagation**
- **Search**

Before going in more details with these two concepts, it's good to know that CSP problems may not have a solution since some of them can be NP-Hard[1].

Constraint propagation is a technique that permit to perform better the search operation by removing values that cannot be part of the solution. We can achieve constraint propagation in several ways, on top of that we can mention:

- ***Node consistency***
  - Every constraint on a variable must be satisfied by all values in the domain of that variable. So, all the domain can be reduced with only the values that respect all the constraints.
- ***Path consistency***
  - It is more complex than arc consistency because considers two pairs of variable instead of one. Is a way to generalized arc consistency in a path from one node to another.

[1]In computational complexity theory, NP-hardness (non-deterministic polynomial-time hardness) is the defining property of a class of problems that have no solution in polynomial-time

- ***Arc consistency***
  - The main purpose of arc consistency is to eliminate values from the domains of the variable. To do this, we consider two variables and the relation (arc) between them: if there aren't values that respect all the constraints in the second variable's domain for a particular value x, present in the first variable's domain, then we can remove x from the domain of the first variable. We can formalize arc consistency with this definition: a Directed arc (v_i, v_j) is arc consistent if $\forall x \in D_i, \exists y \in D_j$ such that (x, y) is allowed by the constraint on the arc.

Most of the Sudoku puzzle that can be found in the magazine can be resolved only applying these rules until each variable has only one value in their domains.

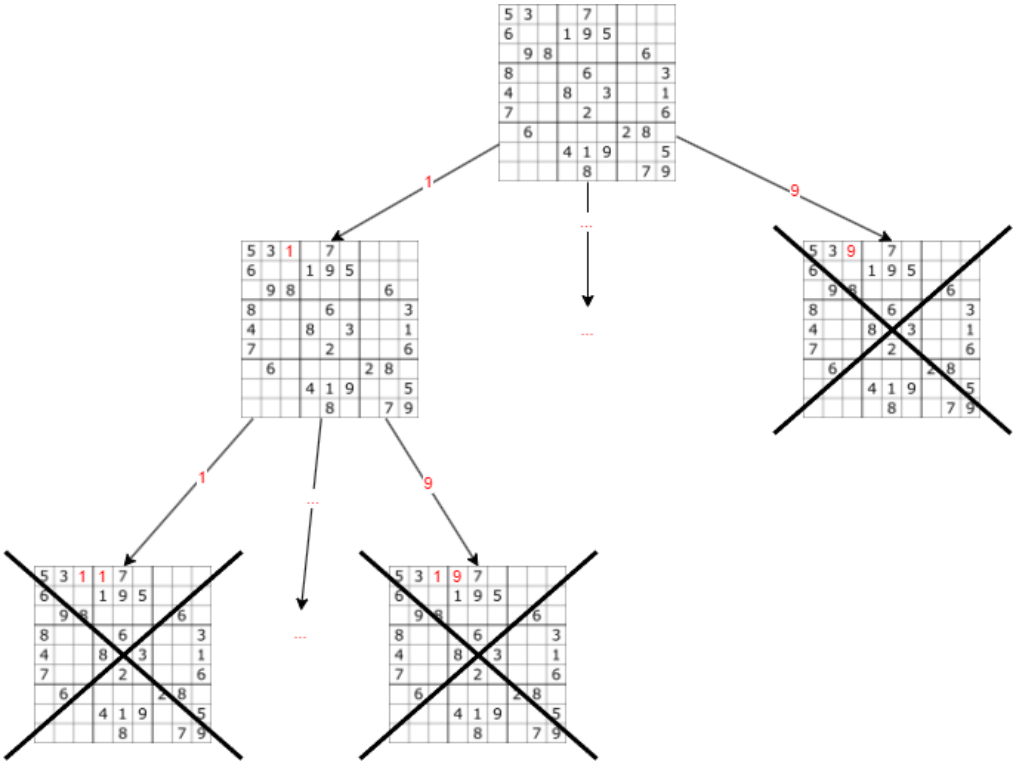
Search algorithms are fundamental to define how we can explore all the possible values for our variables. The most used technique for searching are variants of:

- Backtracking
- Local search

2.0.2 Backtracking

Pure backtracking can be seen as a depth-first-search in a Backtracking-tree. The BT-tree is build with all the possible scenario for our problem that respects the constraint. In our case, starting from a state in which several numbers are already inserted, the next possible states are 9: one for each possible number in the first empty cell. Let's see an example:

Figura 2.1: Backtracking tree

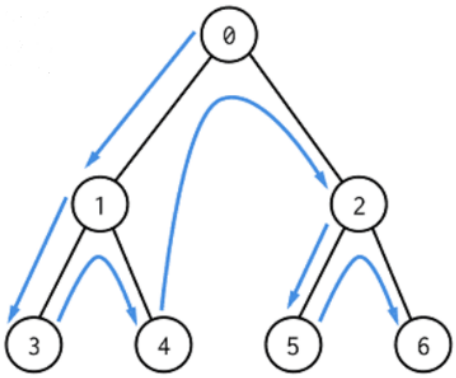


(a) *Example of BT-tree for Sudoku puzzle*

We can also say that the tree branching factor $b = 9$ and the height of the tree is n where n represents the number of empty cell in the starting state.

The depth-first-search is a classic algorithm that permit to explore all the nodes in a tree, following as far as possible every branch of the tree starting from the left one.

Figura 2.2: Depth First Search



(a) *Order of tree examination*

We can look also at some properties of this search:

- **Not complete**[2]: if the tree has an infinite branch the algorithm may not finish even if there was a solution.
- **Not optimal**[3]: In the case the problem has more than on solution, the algorithm may not find the one at minimal depth.
- **Time complexity**[4]: $O(b^d)$ where b is the branching factor and d is the depth in which our goal is situated.
- **Space complexity**[5]:$O(db)$.

2.0.3 Local search

A different approach to handle CSP is to formalize them as optimization problems. The main purpose of optimization problems is to minimize or maximize a given

[2]An algorithm is said to be complete if it finds a solution if one exists

[3]An algorithm is said to be optimal when it will find the best solution if more than one exist

[4]Time complexity is a measure that quantify the time use to perform the algorithm

[5]Space complexity is a measure that quantify the space, in terms of memory, used to perform the algorithm

function respecting all the constraints defined (if there are):

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, i = 1, ..., m \\ & h_j(x) \leq 0, j = 1, ..., p \\ & x \in \Omega \end{aligned} \tag{2.1}$$

Where:

- $f : R^n \to R$ is the **objective function** to be minimized (or maximized) over the n-variable vector x
- $g_i(x) \leq 0$ are the so-called **inequality constraints**
- $h_i(x) \leq 0$ are the so-called **equality constraints**
- $m \geq 0$ and $n \geq 0$
  - If $m = p = 0$, the problem is an *unconstrained optimization problem.*

We can divide optimization problem in several categories:

- *Constraint Linear Problems*
- *Unconstrained Linear Problems*
- *Constraint Nonlinear Problems*
- *Unconstrained Nonlinear Problems*

An example of how we can transform a CSP in an optimization problem is to create a positive objective function $f(x)$ that is $0 \iff$ all contraints are satisfied. As you can imagine, when we find the minimum of the function, we reach our goal. In the Sudoku, we can create a function that can emulate this behavior (we will use it later, in simulated annealing):

$$f(x) = \text{number of empty cell}$$

So when our function reach 0 we can consider our puzzle complete (Obviously when we insert a value in an empty cell we verify that all constraints are respected).

Local search is a technique to find an optimum by performing local minimization

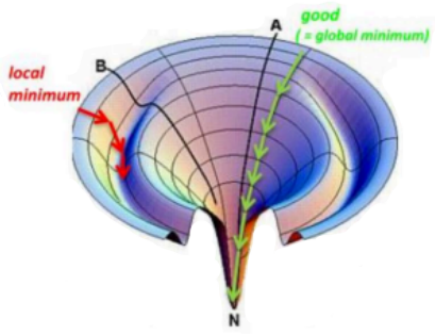
or maximization. The algorithm moves from one state to another, trying to decrease or increase our objective function. To do this, we need to know the neighbor states and the strategy that define in which state we will move.

Hill-climbing is the simplest approach. The name explain the main concepts behind the algorithm: from our starting point, we will move to the nearest hill (maximum[6]). In the terms of local search, we can say that hill-climbing strategies are based on:

- Always move to the adjacent state that minimizes (maximizes) the objective function;
- Terminate if adjacent states do not improve on current state.

One of the main problem in Hill-climbing is the problem of **local minima**: the algorithm could return a value that minimize only locally a function, even if we want to reach the global minimum (or maximum) of that problem. The choice of the starting point became really relevant because the "nearest hill" could be a local or a global maximum.

Figura 2.3: Problem of local minima



There exists several ways to try to overcome this problem and **converge the solution** to a global maximum (or minimum):

[6]Every problem of maximum can be thought as a minimum problem: $max(f(x)) = min(-f(x))$

- restart with different starting points
- allow, on occasion, to make increasing (decreasing) moves;
- Introduce random jumps to reach other position behind the "hill"
- ...

Some famous approach for approximating the global minimum (maximum) of a given function f are:

- For discrete values:
  - Simulated Annealing
  - Local Beam Search
  - Genetic Algorithms
- For continuous values
  - Gradient descent
  - Newton Method
  - ...

In the chapter 4 we will see more in detail the simulated annealing and the algorithm to use it in our Sudoku problem.

Capitolo 3

Backtracking with Constraint Propagation

One way to improve backtracking is to use also the constraint propagation. With this technique we can remove from the BT-tree all the states that result inconsistent with current hypothesis, in this way we can obtain an algorithm that perform better.
But how much propagation we will have to do? We can entrust to the so-called **Forward checking**: we perform constraint propagation only locally, considering domains with unique assignment.
In our case, we can create a second matrix in which we save, for each cell, the corresponding domain. The possible scenario are:

- The **start**: at the beginning we need to initialize our matrix, creating a ***set***, for each cell, with the possible values that can be inserted.
- A **value is inserted**: when we insert a value in one of our cell we create a **copy of our domains' matrix**, and we reduce (constraint propagation) the domain of the cells which are present in the same row, column, and square 3×3. The copy is necessary because if the value chosen was not correct, we need to perform the backtracking, returning to the previous state in which the value of the cell was 0 and the domains were not been modified.
- We **cannot insert any value**: in this case, which corresponds to try to insert a value in a cell with an **empty domain**, we need to perform the **backtracking** because the current Sudoku has no solution.
- When all the domains have the length equal to 1, or when there are **no more empty cells**, our Sudoku is solved.

So let's see the main procedure to resolve the Sudoku (the code of "*isSolved*" and "*UpdateDomains*" method is not provided in this paper, but their implementation, as you can imagine, is very simple). The domains' matrix passed on the first call of "*solveSudoku*" has been already initialized.

```
def solveSudoku(sudoku, domains):
    if isSolved(sudoku):
        return True
    row, col = nextCell(sudoku, domains)
    for i in domains[row][col]:
        sudoku[row][col] = i
        newDomains = UpdateDomains(row, col, i, copy.deepcopy(domains))
        if solveSudoku(sudoku, newDomains):
            return True
        else:
            sudoku[row][col] = 0
    return False
```

The algorithm can be improved by choosing an order as the search proceeds. We can do that through the so-called rule: **Most Constrained Variable**[1]. The "*nextCell*" method is implemented following this idea: the selected cell will be the one with the ***smallest domain*** among all the empty cells.

A simple example that shows the advantages of using this approach is the following: if the algorithm chooses a wrong value for a cell leading to a Sudoku with no solution, the "*nextCell*" method will select the indexes of the cell with the empty domain, and the function "*solveSudoku*" will return immediately **False**.

In terms of complexity, this algorithm performs as follows:

- **Temporal complexity**: $O(9^n)$ where n is the number of empty cell.
- **Spatial complexity**: $O(1)$ because we are not creating the all BT-tree, but we are exploring using forward-checking.

[1]Pick variable with the fewest legal values to assign next, in this way we minimize the branching factor

Capitolo 4

Simulated Annealing

Simulated annealing, as we discuss before, is a **probabilistic technique** for approximating the global minimum (maximum) of a given function f. The name of this method comes from a metallurgic technique that provide a controlled cooling system. The simulated annealing heuristic states that given the next possible state s from the current state s, we *probabilistically decide between moving the system to state s or staying in-state s.* This probability depends on some parameter called ***Energy*** and ***Temperature***.

- ***Energy***: is the value of our objective function evaluated in the states we are considering, in particular $e = f(s)$ and $e^* = f(s^*)$
- ***Temperature***: A time-varying parameter

Given the probability $P(e, e^*, T)$ the main concept behind simulated annealing is that

$$\begin{aligned} T \to 0 &\implies P(e, e^*, T) \to 0, \quad \text{if } e^* > e \\ T \to 0 &\implies P(e, e^*, T) > 0, \quad \textit{otherwise} \end{aligned} \tag{4.1}$$

So, knowing that simulated annealing is an approach to solve optimization problems we can say that 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. The main principle behind this approach is the following:

If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1

Let's see how we can solve the Sudoku using this strategy:

```
def simulated_annealing(f , initial):
    current = initial
    for T in schedule(20000):
        if T == 0 or isSolved(current):
            return current
        next = random_select(neighbor(current, initial))
        de = f(next) - f(current)
        if de < 0:
            current = next
        else :
            if random.uniform(0,1) < math.exp(- de/T):
                current = next
    return current
```

In simulated annealing is really important to choose:

- ***Function*** we need to minimize.
- Which are the ***states*** of the system and how we can pass from one to another, in order to define the neighbors.
- How we create the ***set of temperatures*** and how slowly we decrease them.

4.0.1 Objective function

As we said in the previous chapter, a good function that represents the Sudoku puzzle as an optimization problem is the following:

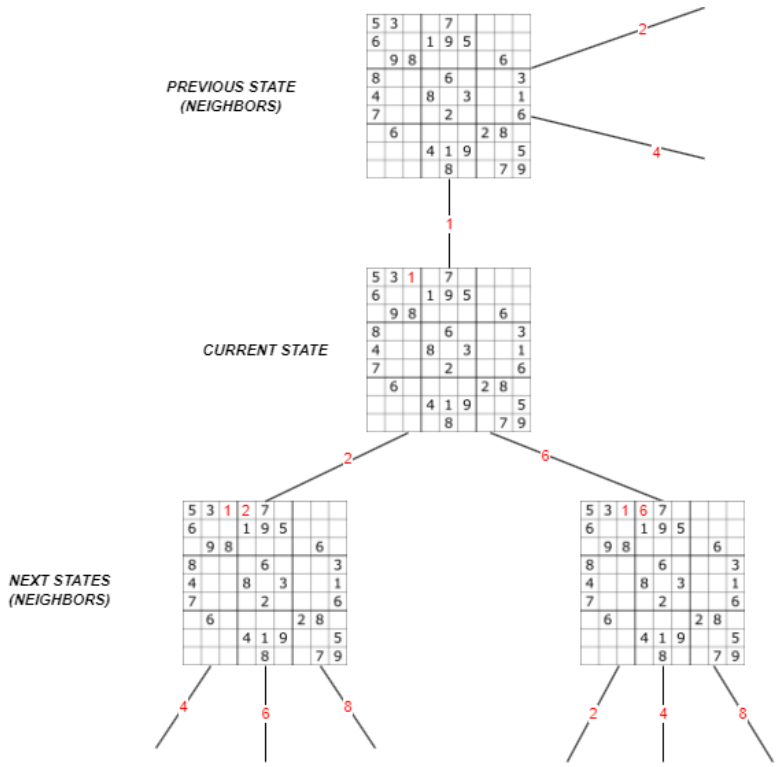
$$f(x) = \text{number of empty cell}$$

As you can imagine, our purpose is to minimize the objective function.

4.0.2 States and neighbors

Let's look at the picture 4.1: we decide that from a starting state, the next possible states are obtained by inserting all the possible values (the one respecting all the constraints) in the **next empty cell**.

Figura 4.1: States and neighbors



(a)

So, we can define the set of neighbors of a state as:

- The list of Sudoku with the next empty cell filled with the possible values
  - In this case, the objective function returns a lower value than the current state or, in other words, the energy produced by these states is lower.
- The previous Sudoku state in which the least inserted value is removed

– In this case the objective function return a higher value than current state, in other words, if this Sudoku is the one extracted from the list of all the neighbors, the Temperature will determine if our system will move or not to this state.

4.0.3 Temperatures

The list of temperatures is created through the method "***schedule***" that receives a parameter n:

```
def schedule(n):
    initial_temperature = 10
    T = initial_temperature
    L = []
    for i in range(n):
        L.append(T)
        T = T - initial_temperature/n
        if T < 0.0:
            T = 0.0
    return L
```

As you can see, the decreased temperature follow a very simple law:

$$T_{t+1} = T_t - T_{initial}/n$$

and the speed with the temperature decrease is proportional to parameter n[1].

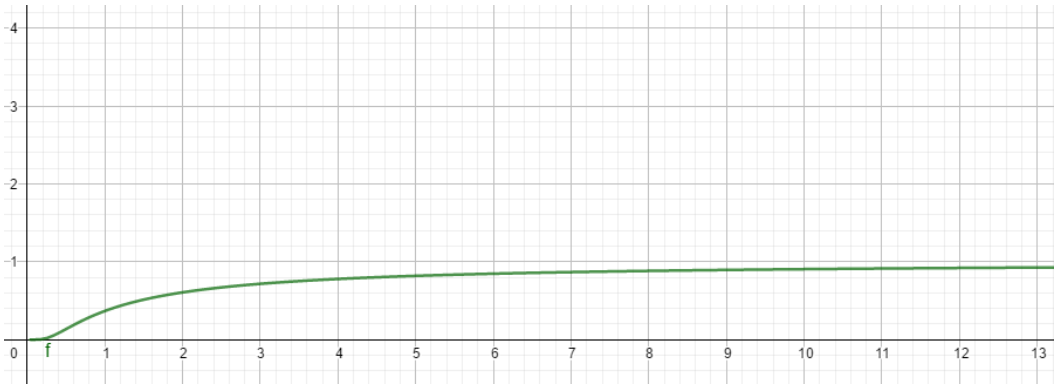
This will lead to a correct behavior: if we are near the initials state, the temperature is very high and the probability to move to a state with energy higher than current state is almost 1 (as the $e^{-de/T}$ formula states), in fact at the first steps our Sudoku could be filled with erroneous value. When we are near the final state and the temperature is going down the probability to move to a previous situation will tend to zero, this is what we want because in the final steps our Sudoku will have many rows, columns, or grid 3×3 with only one missing number, so the probability of inserting a wrong number is very low.

[1] we use a parameter n and not a constant value that slow enough the temperature, as for example, 50000, because, later on, we want to discuss how the decreasing rate affect the performance of the algorithm

In our algorithm the objective function count the number of empty cell: so the only possible case in which the energy difference is greater or equal than zero is when we remove the last inserted number, in other words, $de = 1$ and the probability of moving to the previous state is generated by the formula (figure 4.2 is the associated graph):

$$P = f(T) = e^{-(1/T)}$$

Figura 4.2: Exponential graph



From the graphs 4.2 we can understand why we choose the initial temperature as ten: choosing a higher value is meaningless because the probability will always tend to one: if we start with 1000 and decrease with another law, probably the first 400 probability computed will be, more or less, one. The right choice is a lower value for initial temperature (10 in this case because, as we will see later, this algorithm falls on a lot of local minima) with a correct law for decreasing the values.

One last thing to say is that we could increase the Temperature when we notice that the problem is stuck in a local minimum to improve the general algorithm but, for simplicity, this idea has not been implemented in the source code.

Now, let's test the efficacy of simulated annealing using different n and observe what solutions we will reach (obviously the system is not deterministic, if we run several times with the same n we can get different results because the element extract from the list of neighbors is picked randomly).

Our starting Sudoku is the following:

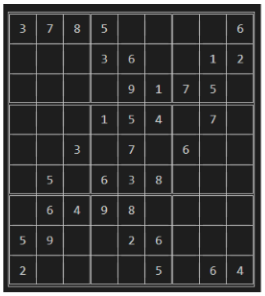
Figura 4.3: Initial Sudoku

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

and the resulting Sudoku puzzles based on the velocity of decrease:

Figura 4.4: Decreasing rate

n = 10



3	7	8	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

n = 100

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

n = 1000

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

n = 10000

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

4.0.4 Limits and another approach

The algorithm created with this idea of neighbors and objective function has some very ***critical limits***:

- In the neighbors, we are only considering the next empty cell (local search), otherwise, if we consider all the empty cell then we get very high space complexity.
- The idea behind the objective functions and the local search of neighbors lead us to many situations where we get ***stuck on local minimum***.

How we can overcome this problem? We use a different approach of neighbors and minimization function that can fit better our problem.

So let's now consider this "*improved*" simulated annealing algorithms, where:

- $f(x)$ = numbers of errors for each row and column
  - This can be computed counting how many different values are present in each row and column (maximum $162 = 9 \times 9 + 9 \times 9$).
  - To find the minimization function, we define the starting values 162, and then we subtract the value computed before. When we reach 0 (global minimum) the Sudoku is solved.
- Sudoku is initially filled with random values that respect the constraint on the 3×3 grid.
- ***Neighbors*** are all the possible Sudoku in which two values in a 3×3 square are swapped (these values are not the one present in the initial state)
  - If we choose a random square and swap two random values, we achieve the same results as selecting a random Sudoku from a list with all the possible neighbors. In this way, we avoid big computational space.

Using the same structure of the code seen before (also the same method for temperature) and choosing a small (as we see before) initial temperature of 0.5 (after same tests can be notice that with this value the algorithm perform better) we achieve a better solution for our problem.

Capitolo 5

Conclusion

One last evaluation that we can make is: measure the performance, in terms of elapsed time and resolved Sudoku, on both the algorithm using several generated Sudoku ($N_{test} = 100$). For our simulated annealing with use $n = 50000$ as decreasing rate. For the generated Sudoku, we use the library: *dokusan*[1]
To generate a new Sudoku:

```
from dokusan import generators
sudoku = generators.random_sudoku(avg_rank=150)
```

avg_{rank} option roughly defines the difficulty of the Sudoku.

Performance			
Algorithm	avg_{rank}	Average time	Solution found
BT with CP	70	0.1051235104 ms	100%
BT with CP	150	0.8787019587 ms	100%
Simulated Ann.	70	0.9525252331 ms	43.0%
Simulated Ann.	150	1.6052895955 ms	7.0%
Improved SA	70	1.2612971853 ms	88.0%
Improved SA	150	3.4237659752 ms	40.0%

As we can expect the BT performs better and find a solution for every Sudoku. In fact, simulated annealing try to approximate the global minimum of the problem and if the temperature not decrease slowly enough for the current Sudoku there is no guarantee that it will find the solution.

[1]Official library documentation: https://pypi.org/project/dokusan/