Artificial Intelligence: knowledge representation and planning

Year: 2017/2018

Assignment 1

Sudoku Solver

Author: **Antonio Emanuele Cinà**

21st March 2018

Index

1 Introduction

A **Sudoku** puzzle is a grid of 9x9 squares. The goal of this game is to assign to each square i a digit from 1 to 9 such that for each block j on the same row, column and 3x3 box i and j are pairwise different. The puzzle is given, possibly, with some filled squares and the player has to complete the rest of squares such that game rules are satisfied. An example of possible starting puzzle and its solution is proposed on the following:

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

Figure 1: Sudoku puzzle

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

Figure 2: Sudoku puzzle solution

A sudoku puzzle can be solved with a unique solution or possibly with multiple solutions, depending on its starting configuration, which determines also its complexity. In both of the two cases a good strategy for the player is to reduce the number of guesses taking advantage of the existing constraints given by game rules. Solving a sudoku can be viewed as a search problem in which the states correspond to partially filled in puzzles.
A Sudoku puzzle can generates 4×10^{38} possible states, so a brute force approach to solve it is not efficient in terms of time and space. It is necessary to reduce this large space without searching in states for which is not possible to reach the solution. One basic solution consists of taking advantage from the constraints, rules, imposed by the game that will be discussed on the next section. We don't have to search for all 4×10^{38} states because when we explore a new state we can immediately eliminate many other states which are not consistent with game rules. Following this idea we can say that Sudoku can be seen as a particular problem in which game rules limit the search space, allowing to develop efficient algorithms for solving it.

2 Constraints

The Sudoku problem can be modelled as a **Constraint Satisfaction Problem** (CSP) which consists of:

- ◇ A set of variables $X = x_1, ..., x_n$.
- ◇ $\forall x_i$ is associated a domain set D_i which specifies the possible values that the variable x_i can assume.
- ◇ A set of constraints which can modify the domain set of each variable.

In particular in case of the Sudoku, variables are represented by each cell on the game board (81 variables) and the initial domain set of each variable is composed by the range of values starting from 1 to 9. Constraints are given by game rules and can be subdivided in three categories:

- ◇ **Row constraint**, values on the same row have to be distinct. Given the cell x_{ij} filled with value $\lambda \in [1, 9]$ the row constraint consists to verify that

$$\forall k \quad x_{ik} \neq \lambda \quad k \in [1, 9]$$

- ◇ **Column constraint**, values in the same column have to be distinct. Given the cell x_{ij} filled with value λ the column constraint consists to verify that

$$\forall k \quad x_{kj} \neq \lambda \quad k \in [1, 9]$$

- ◇ **Box constraint**, values in the same block (3×3 square) have to be distinct. Given the cell x_{ij} filled with value λ the box constraint consists to verify that

$$\forall z, k \quad x_{zk} \neq \lambda \quad z \in [i, i+3] \quad k \in [j, j+3]$$

  For simplicity the previous formal definition is written considering the squares on the top left of each box, but the idea is that each box contains the permutation of values from 1 to 9.

For each cell so there are 20 constraints, 8 given by row constraint, 8 given by column constraint and 4 by block constraint. The following example takes in consideration the tile $(5, 2)$. The blue tiles are affected by row constraint, the red ones are affected by column constraint and the green one by block constraint.

Figure 3: Sudoku constraints

Another interesting constraint is informally called **indirect constraint** which consists to take advantage of filled cells to restrict if it is possible the domain set of white cells. In order to be more clear the following example is considered:

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

Figure 4: Indirect example

On this example we can see that the cell x_{12} has not a direct value, so the first idea could be to try of guessing the right value to put there. But if we consider also the assigned values on the rest of the cells we can observe that:

- ◇ In the column 1 the value 5 is just positioned, so we can exclude to assign value 5 inside positions x_{11} and x_{21}.
- ◇ Inside the second and third row the value 5 is just positioned, and for this reason cells x_{22}, x_{32} and x_{33} cannot assume value 5.
- ◇ Box 1 has to contain value 5 for the satisfaction of box constraint.

From the intersection of these 3 results we can see that the unique and available cell that can contain value 5 is x_{12}. As we will see later this indirect constraint improves a lot the performance of the searching algorithm since it reduces the number of guess, that can bring to incorrect solutions.

In general constraint problems include *NP-Hard* problems, for which we know that the time complexity of the algorithm increases exponentially with the problem size. For instance if we want to solve sudoku with a brute force strategy we should try all the possible configurations and find the one that satisfies the constraints starting from the initial configuration. Considering n as the number of cells 81 and m as the cardinality of the initial domain, the time complexity of this algorithm is $O(m^n)$.
A solution for **Constraint Satisfaction Problem** consists of assigning to each variable a value from its domain, in such a way that every constraint is satisfied. We may want to find:

- ◇ all the possible solutions.
- ◇ only one solution.
- ◇ the optimal solution.

For our purpose I decided to stop the algorithm when a solution is reached, but with few and simple changes of conditions the designed algorithm can work also to find all the possible solutions.

3 Constraint Propagation and Backtracking Algorithm

Instead of try all the possible combinations, that will spend a lot of computational time, the implemented algorithm adopts **constraint propagation** strategy. The main idea is to remove values that cannot be part of the final solution. For sudoku game the algorithm removes the values that will not satisfy constraint properties of the game. In general each cell is associated to an so called **domain set** which specifies the possible values that can be filled inside it. When a cell is filled with a new value i the algorithm propagates the constraints removing value i from the domain sets interested from rules of the Sudoku game.
Some Sudoku puzzles in general can be solved easily with simple constraints propagation, bringing to the final solution very fast. But in some cases, with more complex configurations, it is necessary to make a guess and verify if it brings to the final solution or not. So in case of uncertainty the algorithm has to guess the possible right value, but what if the guess does not bring to the final solution? In this case it is necessary to come back to the guessing state and repeat with another available guess. This idea is basically implemented by **backtracking** strategy.

Backtracking is used for problems that generate a large state space tree. A state space tree is a tree in which each node represents a possible state of the problem. Backtracking traverses this tree in *depth-first order* for finding the solutions of the given problem. If at some step it becomes clear that the current path that cannot lead to a solution it goes back to the previous step (**backtrack**) and chooses a different path.

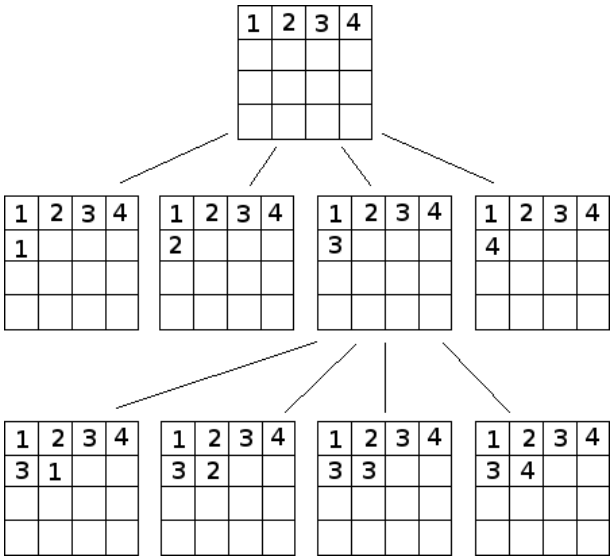

Figure 5: Sudoku 2x2 Backtracking State Space Tree

A Backtracking algorithm is very simple to implement and ensure to find all the possible solutions, since it performs a complete search inside the state space three. But it has some important drawbacks:

- ◇ Search in different branches of the space can bring to reach the same failure states.
- ◇ With the increasing the number of branches from one state cause the decreasing in performance.
- ◇ Each state has to be stored inside a stack or design a recursive algorithm but both of the cases require a large amount of space.

Backtracking has also large time and space complexity since basically it explores all the possible states. How to avoid this? A good solution is to combine **backtracking and constraint propagation algorithm**, so that constraint propagation reduces the number of explored states pruning inconsistent branches composed by states that do not satisfy constraints. Starting from an initial state the algorithm try to explore only the states that satisfy the constraints of sudoku. If the branch does not bring to the final solution the algorithm comes back to a consistent state and search in another branch.
Also in this case the complexity of the algorithm depends on the number of guess that the algorithm performs, so for this reason *indirect constraints* are very useful because they reduce the number of guesses.

3.1 Implementation

On this section we are going to explain the general behavior of designed algorithm and what kind of strategies were adopted to solve as better as possible a Sudoku puzzle.

```
# Sudoku solver backtracking and constraint propagation
isFinish = False
# stack of visited states before guessing
StatesStack = list()
board = readSudoku(filename)
current_state = State(board)

while not isFinish:
    if not AssignQueue.empty():
      value = AssignQueue.pop() # there is a cell to fill
      current_state.assignValue(value)
    else:
      # try to apply possible indirect constraints
      # in order to reduce guessing
      indirect = current_state.indirectConstraint()
      if not(indirect):
        # pick index with short size
        x, y = current_state.indexMinDomain()
        if x== -1 and y == -1:
          # no available cell for guessing
          if not (len(StatesStack) == 0):
            # back to previous state
            current_state = StatesStack.pop()
          else:
            raise Exception("No possible solution are discovered")
        else:
          # if the domain set has only one value don't guess
          # it but assigns it directly
          if current_state.isUnique(x, y):
            val = current_state.getDomainSet(x, y)[0]
            current_state.assignValue(Value(val, x, y))
          else:
            # ambiguity: guess and push state into state stack
            guess = current_state.guess(x,y)
```

```
35              copy_state = copy.deepcopy(current_state)
                StatesStack.append(copy_state)
37              current_state.assignValue(guess)
  isFinish = current_state.isFinish()
39 current_state.print()
```

First the algorithm reads, from a csv file, the starting configuration of the Sudoku puzzle, and meanwhile it creates a 9x9 matrix representing the cells of the puzzle and their domain set. Initially each domain set has values from 1 to 9.
Considering the domain set of a generic cell:

- ◇ it becomes empty when the corresponding cell is filled with a value.
- ◇ some elements can be removed from it as a consequence of constraint propagation effects.

On the following example we can see that some cells are just filled with values $2, 7, 1, 3$ and their domains set are empty, meaning that they can't change value until the end. When a new value i is assigned propagation values happens, removing from the cells on the corresponding row, column and block i from their domain sets.

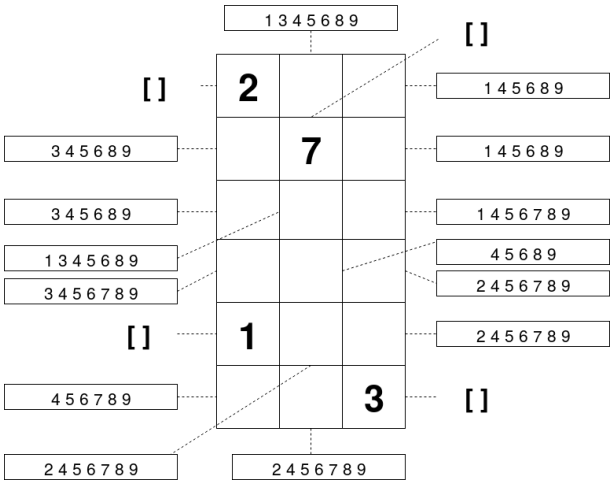

Figure 6: Puzzle composition

During the propagation phase of direct and indirect constraints the algorithm can recognize that some cell can assume only an unique value. When a new value is to be positioned the algorithm push into a special queue, called **AssignQueue**, the assignment event, for which specifies coordinates position (x, y) and digit. Since during propagation phase there can be discovered multiple values that can be directly assigned. With this implementation the starting phase is equal to the rest of the algorithm, during the reading phase if the current character is different then zero it creates the assignment event and push it inside the queue. When the queue is empty it means that with only direct constraint there is ambiguity and for this reason it is necessary to guess. Since the complexity of the algorithm increases with the number of guess, develop a good strategy to reduce them is fundamental. A first optimization is to apply *indirect constraints*, removing values that for sure have a fixed position. If no indirect constraint is applied, the algorithm takes the coordinates of the cell with minimum length of the domain set. This last strategy is called **most constrained variable**, which consists on choosing an unassigned cell with the fewest consistent values left in its domain set. In general backtracking

uses fixed ordering of variables, but this is not always efficient. MCV is a simple heuristic that changes dynamically the order of search proceed and with a certain probability reduces the number of wrong guess. The idea is to consider the domain set with minimum length so that the probability of making a wrong guess is lower. Suppose of considering the cell at position i with domain set composed by 7 possibilities $(1, 2, 5, 6, 7, 8, 9)$. The probability to make a wrong guess is $\frac{6}{7} \approx 0.857$. If instead we consider the cell j, which has only has 2 possibilities $(8, 9)$ the probability to make a wrong guess is $\frac{1}{2} = 0.5$, meaning that it is convenient to choose j respect to i.
In case of guess there is not certainty that the guessed value is right, meaning that it probably can bring to an unsolvable state. For this reason it is necessary to put in a state stack, called **StateStack**, the current state before making the guess so that in case of wrong guess it is possible to go back in a stable state and searching for another guess. This behavior simulates what we have described before as the **backtrack step** of a backtracking algorithm. So in case of multiple choices at cell i the algorithm removes the guessed value x from the cell i and push the state into the stack. The removing step of the values x from the domain set is important to avoid that later on the same state the algorithm will pick again x.
For each iteration the algorithm checks if the board is complete, simply verifying if 81 tiles are filled correctly.

3.2 Performance

In this section we are going to discuss the performance given by the implemented algorithm. The measures that are considered are so described:

- ◇ rating, complexity of the sudoku puzzle;
- ◇ ndirect, number of direct constraints applied;
- ◇ nindirect, number of indirect constraints applied;
- ◇ nunique, number of unique choices;
- ◇ nguess, number of guess;
- ◇ nback, number of backtrack;

Note that time is not considered since it depends also from the machine in which the algorithm is executed. On the following are reported 10 values, each of which is computed as a mean of 4 runs for each puzzle.

rating	ndirect	nindirect	nunique	nguess	nback
1	145	0	0	0	0
2	264.25	0	2.25	3.75	2.25
3	147.25	0	0.75	1	0.75
4	1025.25	0	28	29.25	28
5	2541	0	92	94.75	92
6	1846.5	0	50.5	51.75	50.5
7	1179	0	49	53.75	49
8	1985	0	58	60.25	58
9	2004.5	0	59	61	59
10	259459.75	0	11462.5	11629.5	11623.5

Table 1: CPB algorithm results without indirect constraints

rating	ndirect	nindirect	nunique	nguess	nback
1	145	0	0	0	0
2	112	1	0	0	0
3	88	2	0	0	0
4	139.5	8	0.5	1	0.5
5	88	5	0	0	0
6	100	3	0	0	0
7	285	11.75	2.25	5.75	3
8	417.75	17	2	7	5.75
9	757	22.25	4.75	11.5	9.25
10	53952.8	2297	353.8	1497.6	1495.8

Table 2: CPB algorithm results

The two tables summarize the behavior of the algorithm considering the evaluation of only direct constraints and with also indirect constraints. What we can see is that indirect constraints reduce significantly the number of guess, meaning that the algorithm is faster on finding the solution. Another interesting evaluation regards the number of backtrack, they are larger when indirect constraints are not applied and small with them. The reason is that indirect constraints reduce a lot the size of the domain sets and possible guesses have a greater probability to be right. In general the greater is the complexity/rating of the puzzle the greater is the number of guess required by the algorithm, with the consequence of wasting time and resources with wrong guesses. This effect is given by backtracking strategy, which guarantee completeness and if there is a solution it will find it but the drawback is that it wastes time in branches that are not useful. For that reason adding constraints improve the performance of the algorithm since the branches are pruned first.

4 Relaxation Labeling Algorithm

Another interesting approach consist of not use only local information but also consider contextual information. In general contextual information can be useful for labeling problems, sometimes local information are not sufficient or ambiguous.
A labeling problem is defined by:

- ◇ A set of n objects $B = \{b_1, b_2, ..., b_n\}$
- ◇ A set of m lab $\Lambda = \{1, ..., m\}$

The goal of this class of problems consists to assign a label from Λ to each object of B.
Contextual information are expressed in terms of $n^2 \times m^2$ matrix of compatibility coefficients:

$$R = r_{ij}(\lambda, \mu)$$

The coefficient $R = r_{ij}(\lambda, \mu)$ measures the strength of compatibility between the two hypotheses "b_i is labeled λ" and "b_j is labeled μ".
From this idea we can imagine sudoku game not only as a constraint satisfaction problem, but also as a labeling problem in which:

- ◇ the set of objects is composed by the 81 cells of the board.
- ◇ the set of labels is composed by the digits $1, ..., 9$.

The compatibility matrix is expressed in terms of game rules, it is a function that return 0 or 1 if the assignment is valid or not. The compatibility function for sudoku game is so defined:

```
def compatibility(i, j, lb, mu):
  if i == j:
    return 0
  if lb != mu:
    return 1
  if inSameRow(i,j) or inSameColumn(i,j) or inSameBlock(i,j):
    return 0
  return 1
```

Note that the compatibility with itself is defined equal to zero, and it is one only if game constraints are satisfied for this assignment. For instance, the assignment $(1,1) = 2$ and $(1,2) = 2$ is not valid, compatibility equal to 0, since row constraint is not satisfied.

A relaxation labeling algorithm start with an initial m-dimensional probability vector for each object $i \in B$:

$$p_i^{(0)}(\lambda) = (p_i^{(0)}(1), ..., p_i^{(0)}(m))^T$$

with $p_i^{(0)}(\lambda) \geq 0$ and $\sum_\lambda p_i^{(0)}(\lambda) = 1$

$p_i^{(0)}(\lambda)$ is the probability distribution associated to the object i at time 0, and represent the probability that the object i is right label with λ. Each object is associated to one probability distribution, and the concatenation of all of these defines a **weighted labeling assignment** $p^{(0)} \in \mathbb{R}^{nm}$. All the possible weighted labeling assignment belong into a space IK so defined:

$$\text{IK} = \Delta^m = \Delta \times ... \times \Delta$$

where each Δ is the standard simplex of $\mathbb{R}^n$.

$$\text{IK} = \left\{ \overline{p} \in \mathbb{R}^{nm} | \quad p_i(\lambda) \geq 0, \quad i = 1, ..., n, \quad \lambda \in \Lambda \quad \text{and} \quad \sum_{\lambda=1}^{m} p_i(\lambda) = 1, \quad i = 1, ..., n \right\}$$

Each vertex of IK represents an unambiguous labeling assignment, that is one which assigns exactly one label to each object. The optimal solution, for a labeling problem, consists exactly to find these points such that there is no ambiguity.
Starting from an initial weighted labeling assignment, the relaxation technique reduces iteratively the ambiguity and disagreement of the initial labels considering the compatibility r_{ij} previous defined between objects and labels. The idea is that at the end of the iterative procedure all the probability distributions have an high probability, ideally equal to 1, to the right label that should be positioned in that cell. Relaxation strategies do not necessarily guarantee convergence, and thus, the procedure may not arrive at a final labeling solution with a unique and unambiguous assignment of the labels for each object and that satisfy game constraints. In fact, at the end of the iterative procedure it can happen that some vectors provide ambiguity or inconsistent assignments. The following figure shows an example of possible weighted labeling assignment at a generic time t:

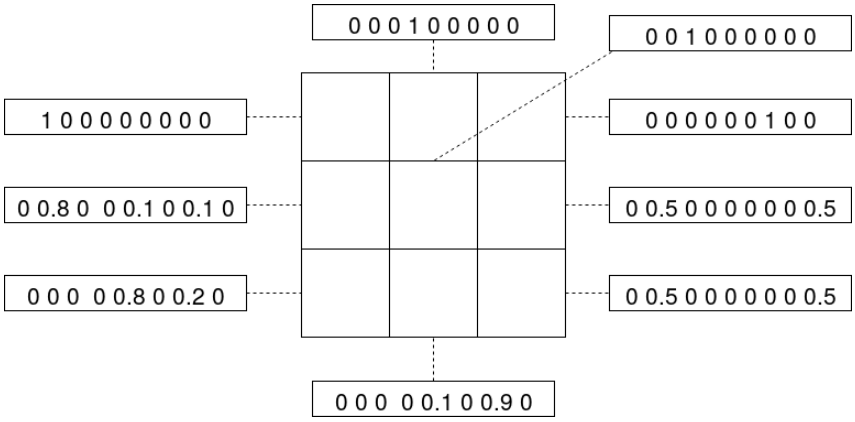

Figure 7: weighted labeling assignment

What is possible to see is that there is no uncertainty for cell $(1,1),(1,2),(1,3),(2,2)$ that will be labeled correspondingly with digits $1,4,7,3$. Instead for the cells $(2,1),(3,1),(3,2)$ there is a slight uncertainty, but not so significant. But the problem regards cells $(2,3),(3,3)$ for which there is total uncertainty for what kind of values should be selected.
Each iteration step updates the vectors probability using the following heuristic formula, provided by Rosenfeld, Hummel and Zucker in 1976:

$$p_i^{(t+1)}(\lambda) = \frac{p_i^{(t)}(\lambda) q_i^{(t)}(\lambda)}{\sum_\mu p_i^{(t)}(\mu) q_i^{(t)}(\mu)}$$

$$q_i^{(t)}(\lambda) = \sum_j \sum_\mu r_{ij}(\lambda,\mu) p_j^{(t)}(\mu)$$

where $q_i^{(t)}(\lambda)$ measures the score associate to the hypothesis "b_i is labeled with λ" at time t.

Another interesting consideration regards the stopping criteria of the algorithm, which evaluates the so called **Average local consistency**:

$$A(p) = \sum_i \sum_\lambda p_i(\lambda) q_i(\lambda) \qquad p \in \mathrm{IK}$$

Marcello Pelillo, in 1997, proved that the relaxation operator increases the average local consistency on each iteration, meaning that:

$$A(p^{(t+1)}) > A(p^{(t)}) \qquad \forall t = 0,1,...\text{until a fixed point is reached}$$

So the stopping criteria has to consider the convergence of the algorithm. For this reason the following criteria is considered:

$$\text{step}^{(t+1)} = A(p^{(t+1)}) + A(p^{(t)})$$

When the **step** measure is lower then a given *threshold* ε the algorithm stops, meaning that it has reached the convergence.

$$\text{if} \quad \text{step}^{(t+1)} > \varepsilon \quad \text{then} \qquad \varepsilon \in \mathbb{R}$$

then convergence is reached

Relaxation labeling strategy has some drawbacks that should be considered:

- ◇ The goal of a sudoku solver is to find a solution, which is consistent with game rules and constraints. Game rules define a relation between all the cell of the board. In that case, the support for assignment of label to a cell can be very high if it is consistent with most of the rest cells, even if it is not consistent with all of them. But for the rules of sudoku this is not consistent, and it can bring to a wrong solution.
- ◇ Many possibilities for equal values and relaxation labeling algorithm does not handle this problem.
- ◇ Lacking of a formal definition of the stopping criteria. Finding a solution does not mean always reach the convergence for the relaxation labeling, so the algorithm will continue to run even if a solution was found or will stop even if the solution is not reached.

4.1 Implementation

For this approach two algorithms was designed, the first one apply directly the formula using iteration and the second one use matrix notation to speed-up the operations. The first presented algorithm use language iterators and apply the defined formula for updating the weighted labeling assignment:

```
def createQ(): # create matrix q of scores
  q = np.zeros((81, 9))
  for i in range(81):
    for lb in range(9):
      for j in range(81):
        for mu in range(9):
          q[i][lb] = q[i][lb] +
            compatibility(i, j, lb, mu) * p[j][mu]
  return q
# update the matrix prepresenting the weighted labeling assignment
def updateP():
  q = computeQ()
  p = p * q
  # normalize vector p since it represent a prob distribution
  row_sums = numerator.sum(axis=1)
  p = p / row_sums[:, np.newaxis]

def averageConsistency():
  return np.sum(p*q)

def relaxationLabeling():
    avg_t = 0
    step = 1
    # stopping threshold defined = 0.001
    while step > 0.001:
      updateP()
      avg_t1 = averageConsistency()
      step = avg_t1 - avg_t
      avg_t = avg_t1
```

The next algorithm apply the same formula but use matrix notation and operators:

```
def relaxationLabeling():
  step = 1
  avg_t = 0

  while step > 0.001:
    q = np.dot(rij, p)
    p = (p * q).reshape(81,9)
    row_sums = p.sum(axis=1)
    p = (p/row_sums[:, np.newaxis]).reshape(729,1)
    avg_t1 = averageConsistency(q)
    step = avg_t1 - avg_t
    avg_t = avg_t1
  p = p.reshape(81, 9)
```

At the end of the two procedures it is necessary to iterate for each cell of the sudoku puzzle and fill the cell with the label that has the larger probability inside the distribution.
One of the problem of relaxation labeling algorithm for solving sudoku is that neighbor cells can bring until the end same the same probability to the same value. Suppose of considering two cells i and j, and the right choice is to assign 7 to i and 4 to j. It can happen that i and j have the same probability for 7 and 4 and incorrectly the algorithm will pick 7 for the two cells. One way to reduce this effect is to add like an oscillation of the probabilities at the initial state, instead of starting with the same probabilities.

4.2 Performance

The two implemented algorithms apply the same formula but the first one is slower than the second one. The reason is given by the fact that the algorithm is developed using Python, an high level language, and iterators have not an efficient implementation. Instead the second one perform simply matrix operation using the library *numpy*, which provides efficient implementation of linear algebra operators and N-dimensional array objects.
The following table summarizes the results obtained from different experiments. In order to test the efficiency and efficiency of relaxation labeling algorithm 3 levels of complexity of the sudoku puzzle are considered:

complexity	n iterations	solved
easy	450	*true*
medium	1347	*false*
hard	1858	*false*

Table 3: RL algorithm results

The limit threshold adopted for the stopping criteria is 0.001 and it is possible to see that the greater is the complexity of the puzzle the more iteration are required to reach convergence. Only easy puzzles are solved by the algorithm, meanwhile medium and hard puzzles give solution which doesn't satisfy game rules.

5 Comparison: CPB and RL algorithms

Until now we have discussed about two different approaches to solve the same problem. Each strategy manages rules and game properties/constraints in different ways having in any case the same goal, which is to find a solution. In order to give a better comparison about the two approach different sudoku puzzles was considered. The final conclusion is that CPB guarantees to find a solution independently of puzzle complexity, instead RL is able only to solve easy puzzle with small rating. When the rating increases it is necessary to make guess and this bring consequences in both of the two algorithms. On the first one making guesses brings the algorithm to waste time for backtrack step with possibly wrong guess. On the second one the problem is more complicated, since the possible labels maintain an high probability. Since the selection of the digit is provided only at the end during the procedure it can happen that game rules are not satisfied. In fact with hard configurations the RL algorithm can provide wrong solutions.
In order to give a summarized comparison between this two approaches the following criteria are considered:

◇ **Completeness**

- CPB: yes, it guarantees to find a solution if it exists whatever is the puzzle complexity. In case of wrong guess it goes back using backtrack strategy.
- RL: no, it solves only easy sudoku.

◇ **Optimality**

- CPB: in general there is no optimal solution but if we want to consider the solution with minimum number of guesses so CPB can provide it.
- RL: no, since it solves only easy sudoku.

◇ **Temporal complexity**

- CPB: $O(m^n)$. In principle backtracking explores all the possible states, but it is limited by constraint propagation which prunes first branches that don't give a solution. A good constraints analysis is essential for optimization purpose of this kind of algorithm.
- RL, each updating step cost $\theta(n^2 \times m^2)$, but the total complexity is not well defined since it is repeated until the stopping criteria is reached. As we have seen the stopping criteria adopted is an heuristic and it has not an upper bound, the converge of the algorithm cannot be computed first.

◇ **Spatial complexity**

- CPB: $O(n)$ is the spatial complexity for the *state stack*, that can contain at most 81 states. The spatial complexity used to store and represent the board game and *domain sets* is equal to $O(n \times m)$.
- RL: If we consider the implementation with matrix notation the spatial complexity is $O(n^2 \times m^2)$, which is given by the dimensionality of R_{ij} matrix. If we consider the compatibility matrix as a function the spatial complexity is reduced to $O(n \times m)$, which is given by the usage of matrix to store p and q.

◇ **Stopping criteria**

- CPB: the algorithm terminates when it finds a solution, or possibly when all the possible solutions are discovered.

- RL: uses an heuristic stopping criteria associated to the convergence of the weighted labeling assignment. This does not always lead the algorithm to reach a consistent solution for a sudoku puzzle.

Where n is the number of cells that compose a sudoku board and m is the number of digits that can be assigned. In general $n = 81$ and $m = 9$.

6 Conclusion

Sudoku game belongs into that class of problems which generates a large state space tree. Brute-forcing technique is not applicable, since for the moment there is no hardware that explores all the 4×10^{38} states quickly. Other strategies can be adopted, which basically deal with game rules and its constraints in different ways. In this assignment are considered two strategies. The first one deals sudoku game as a **Constraint Satisfation** problem and defines an algorithm that solves it using **constraint propagation** and **backtracking** technique in order to provide completeness property. The second one considers sudoku as a **labeling problem**, and tries to take advantage on the contextual information and applies iteratively an heuristic formula to update the current state until a stopping criteria is reached.
As we have seen the first one will always find a solution, if it exists, but time complexity increases with the complexity of the puzzle. A possible solution consists to find good strategies to prune first states and reduce the number of guesses. Here are adopted *direct constraints*, given by game rules, *indirect constraints*, given by a better analysis of the game situation, and *most constrained variable*, which reduces the probability to make a wrong guess.
The relaxation labeling algorithm instead works well if the puzzle is not so complicated, since it is able to solve only easy configurations. If the rating increases the algorithm is not always able to take the right decision and updates on the right way the states. This approach seems to be not so efficient for this kind of problems. Better results can be given by problems in which context is essential and local information are not sufficient enough for *labeling decisions*. Possible examples in which contextual information can be fundamental in order to have precisely and correctly results are: *handwriting interpretation* and *edge detection*.