SUDOKU SOLVER

MUHAMMAD TAJWAR
MATRICULATION No. 888394

March 2021

ARTIFICIAL INTELLIGENCE : KNOWLEDGE REPRESENTATION AND PLANNING

PROFESSOR ANDREA TORSELLO

Contents

1 The Problem Introduction **4**

2 Background **4**

2.1 Constraint Satisfaction Problems 4

2.1.1 Properties of CSP 4

2.2 Constraint Propagation 5

2.3 Constraint Propagation with Forward check 5

2.4 Backtracking 5

2.5 Relaxation Labeling 5

2.5.1 Properties of RL 5

3 Goal **6**

4 Constraint Propagation and Backtracking Algorithm on Sudoku **7**

4.1 Problems with Backtracking 7

4.2 Coding and Implementation 8

5 Relaxation Labeling on Sudoku **11**

5.1 Coding and Implementation 12

6 Comparison between Constraints Propagation Backtracking and Relaxation Labeling **14**

7 Conclusion **15**

Abstract

Sudoku is a worldwide game it is a type of brain game that usually people play it to sharpen their minds, so it a 9x9 board game and in this 9x9 board we have nine 3x3 boxes the game is simple we are given a random number from 1 to 9 with some boxes are empty and some are filled with these random numbers and our goal is to fill all the board cells with numbers but with some constraints

There are many ways to solve these puzzles, in this paper I use two algorithms namely Constraint Propagation and Backtracking, and Relaxation Labeling and there are many other algorithms to solve this puzzle and some of which are AC3, Forward Checking etc, and I am going to use Backtracking and then compare it with the relaxations Labeling and see how these two algorithms works. In this paper I also show that how both the algorithm works and how there are implemented

1 The Problem Introduction

Sudoku is a puzzle game with a 9x9 board it is almost similar to the n-queen problem the idea is that in the whole 9x9 board there must be a unique number from 1 to 9 in each column and each row, moreover, we have 3x3 boxes total 9 boxes and in each 3x3 box there must be unique numbers
There are many algorithms to solve these type of problem and this problem is a Constraints Satisfaction Problem, these type of problems have unique or multiple solutions depending on the puzzle easy, medium or hard puzzle so for the algorithm to be smart it must reduces it step to converge to the solution so in that case we will use Backtracking Method and Relaxation Labeling to solve this problem and now I will briefly describe the definitions of these algorithm and how these algorithms works

2 Background

2.1 Constraint Satisfaction Problems

You can find these types of problems in many fields and area of computer science especially in game theory so basically it is search procedures that do their working on specific constraints
The main goal of constraints satisfaction problem in AI is to discover some state problem meanwhile satisfies given constraints for the problems

2.1.1 Properties of CSP

CSP consist of three modules that is VDC (Variables, Domains, and Constraints) in a given CSP problem there exist variables:

$$V = \{V_1, V_2, V_3, ..., V_n\} \tag{1}$$

Each Variable have its own domain

$$D = \{D_1, D_2, D_3, ..., D_n\} \tag{2}$$

and they follow some constraints

$$C = \{C_1, C_2, C_3, ..., C_n\} \tag{3}$$

in the each constraints consist of 2 values

$$C_n = \{scope, relation\} \tag{4}$$

scope means it consists of a set of variables that participates in constraints whereas relation defines the values that a variable can take.

2.2 Constraint Propagation

Inferences in Constraint Satisfaction problems are the constraint propagation that is how the problem propagates and formulate through for example if you take the map coloring problem you always ended up with the big tree but as you put some restrictions on it the tree will begin to prune and become short finite tree easy to find a solution and this way the constraints propagation works

2.3 Constraint Propagation with Forward check

Forward Checking is based on assigning the variable in such a way that it removes the conflicting value for all the neighbor variables this algorithm saves times and tries to shorten the tree as short as possible

2.4 Backtracking

Backtracking follows the Depth-first search(DFS) algorithm it is one of the problem-solving strategies it mainly uses the brute force approach that is trying out all possible solutions and picks up the desired one. Backtracking does not use as an optimization problem we will discuss more this algorithm in more detail in the coming sections and how this is implemented in sudoku game to solve with the best result.

2.5 Relaxation Labeling

Relaxation Labeling is based on contextual constraints so how it works, it works on a group of methods for assigning labels to the objects in a set so it is mainly used for computer vision, some basic properties are shown below

2.5.1 Properties of RL

The labeling problem is defined as objects

$$B = \big\{b_1, b_2, b_3, ..., b_n\big\} \tag{5}$$

and each objects have a set of labels

$$\Lambda = \big\{1, 2, 3, ..., m\big\} \tag{6}$$

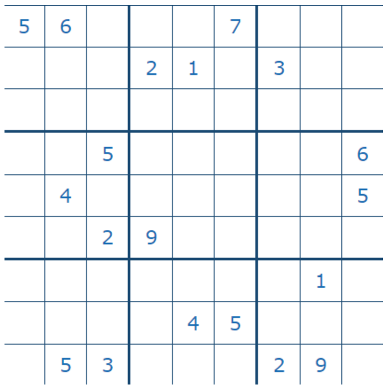
so basically this algo assigns a label from Λ to each object O that is express as matrix of compatibility coefficients of size $n^2 \times m^2$

$$R = r_{ij}(\lambda, \mu) \tag{7}$$

3 Goal

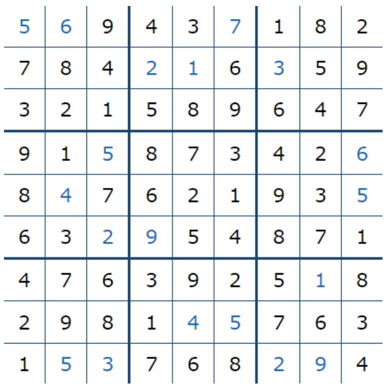
We are given an unsolved Sudoku board initially it would be filled with some digits 1 to 9 and some are left empty so the main goal is to understand the three algorithms which are Constraints Propagation with the forward check, Backtracking, and Relaxation Labeling to solve the Sudoku puzzle

There are many ways to solve the Sudoku puzzle but we have to find the best-optimized algorithm that would solve the solve intelligently with less time. The goal of this game is to assign each square a digit 1 to 9 such that for each block on the same row, column, and box 3x3 would be all different in digits. So the player has to fill all the empty squares such that the rules of the game are satisfied. This is demonstrated in the below pictures



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

Figure 1: Unsolved Puzzle



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

Figure 2: Solved Puzzle

4 Constraint Propagation and Backtracking Algorithm on Sudoku

As aforementioned the basic technique to solve the Sudoku is a brute force technique that is trying all the possible combinations and that will consume a lot of time that we can't afford and this called the Constraint Propagation algorithm. So how this algorithm works? it simply removes the value that can take you to the optimal solution so in this game it removes the value that will not satisfy the constraints domain. Each cell has its domain (2) and which tells you what value could fit in the cell and when the cell has a new value the algorithm starts propagating and remove a value from its domain.

Many puzzles can be easily solved by constraints propagation method but in some cases, Sudoku would be complex and harder puzzle so in this situation we assume a number and try to fit in the cell and verify it that if it takes you to the final round and if not you have to take a step back and repeat this back and forth and this called Backtracking the puzzle.

Backtracking algorithm is not that fast algorithm because of the loops it makes a large state space tree and applies the Depth-first search technique to traverse the tree, the algorithm works simply if at any node it would be clear that this path cannot take you to the final solution than it goes back to the previous step and tries again with new value and a new path.

4.1 Problems with Backtracking

The Backtracking algorithm is very easy to implement as it traverses the whole tree finding all the possible solutions of the puzzle as it follows the depth-first search so it also has some cons.

1. Firstly it creates a large space tree so it takes too much time to transverse the tree thus decreasing in performance
2. Secondly it goes into the depth so many stuck in a loop or goes to infinity

According to the time and space, the constraint propagation behave worst as it finds all the possible solutions and has a large space tree so we have to find the optimized version that can lead to optimal results that is using both the algorithm together Backtracking and Constraint Propagation .

4.2 Coding and Implementation

Now i will show you how the algorithm really works and how the code implements the constraints propagation and backtracking.

```python
def soduko(Sboard):
    print(Sboard) #This shows how the backtracking works
    find = find_empty(Sboard) #finding the first empty cell
    if not find:
        return True #this means we find the solution and we are
    done
    else: #
        row, column = find

    # above code is actually the base case of recursion

    for i in range(1, 10): #Looping thru the board
        if check_validation(Sboard, i, (row, column)):#checking the
     validation of the number
            Sboard[row][column] = i  #if find a valid number insert
     into cell

            if soduko(Sboard):   # this is basic recursive step
    that actually backtracks we keep on trying trying trying to add
                # values untill the solution is find
                return True

            Sboard[row][column] = 0 # if the number is invalid that
     we fit we set the previous value to 0 and again try diff value

    return False

# This function checks the validity of the board
def check_validation(Sboard, number, position):
    #It takes three parameters the Board, the number which we fit
    in the cells, and the position where the number would be fit

    #we are basically checking three things
    #Rows
    #Columns
    #3x3 Boxes


    for i in range(len(Sboard[0])):  # Looping thru the every
    single column of a given row
        if Sboard[position[0]][i] == number and position[1] != i:
    #We check each element in the row we just added in
            #the position[1] != i is that number we just inserted
    in we ignored that
            return False

    for i in range(len(Sboard)): #Now we are checking the column
    vertical
        if Sboard[i][position[1]] == number and position[0] != i:
    # Nothing special its just opposite of row operation
```

```
            return False


    # Now we are checking the 3x3 Boxes in the Board
    # we make sure that No number coincides in the box 1 to 9

    #this 2 codes divides the whole board in 3x3 boxes and assign positions like [0,0] [0,1] [0,2] so on and so forth
    x = position[1] // 3
    y = position[0] // 3



    # looping through the 9 elements
    # Now we making sure that in the boxes now value is appearing twice
    #multiplying by 3 beacuse all the positions are either 0 or 1 2
    for i in range(y * 3, y * 3 + 3): # +3 is for going to the 3rd column and row
        for j in range(x * 3, x * 3 + 3):
            if Sboard[i][j] == number and (i, j) != position:
                return False

    return True
```

Listing 1: Backtracking

The algorithm first take a Sudoku Board as input the board is 9x9 that is 81 cells some cells are filled with the random numbers 1 to 9 and some cells are left empty and we show the algorithm empty cells as ”0” then the next step is to find the cell empty or not

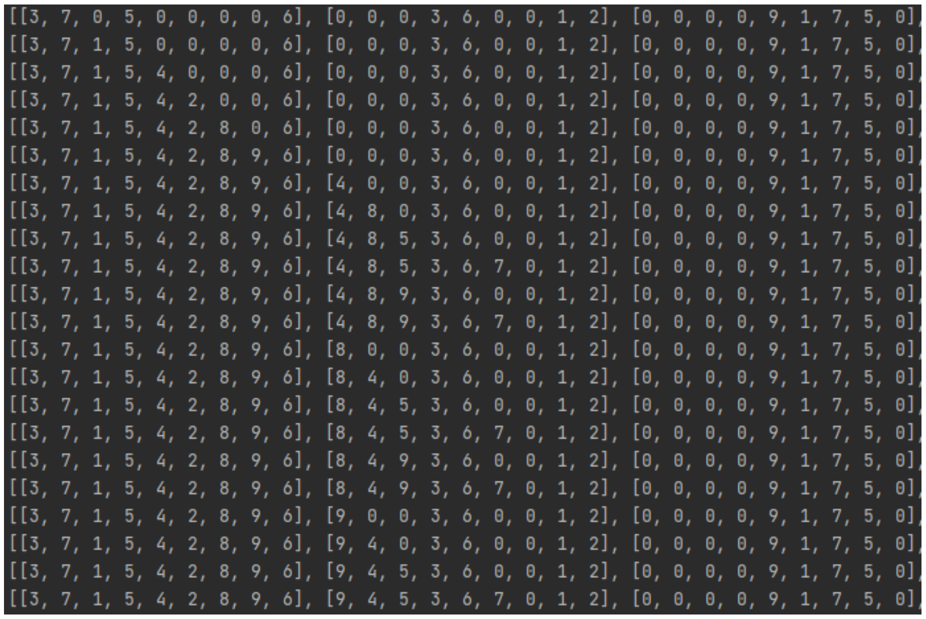
```
def find_empty(SBoard):
    for i in range(len(SBoard)):
        for j in range(len(SBoard[0])):
            if SBoard[i][j] == 0:
                return (i, j)

    return None
```

This code is found empty squares it will loop thru the board where it finds the empty cell it will return the position of that cell

The next step is to check the validation of the Sudoku board the function in Python written as `def check_validation(Sboard,number,position)` do the validation of row,column, and boxes this function makes sure the in each row there would be a unique single number and in each column, there would be a unique single number only from 1 to 9 and in each 3x3 box there would be all different numbers the function check row by row column by column and boxes by boxes each number it will ignore the number that we just inserted in and move to the other cell.

Then finally we use the Backtracking code this code is the basic code that computes all the possible solution but in an optimized way that is it prunes the unwanted branches `def sudoku(Sboard)` takes the board and apply both constraint propagation and backtracking technique and the working of this technique is shown below:



```
[[3, 7, 0, 5, 0, 0, 0, 0, 6], [0, 0, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 0, 0, 0, 0, 6], [0, 0, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 0, 0, 0, 6], [0, 0, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 0, 0, 6], [0, 0, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 0, 6], [0, 0, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [0, 0, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [4, 0, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [4, 8, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [4, 8, 5, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [4, 8, 5, 3, 6, 7, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [4, 8, 9, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [4, 8, 9, 3, 6, 7, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [8, 0, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [8, 4, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [8, 4, 5, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [8, 4, 5, 3, 6, 7, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [8, 4, 9, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [8, 4, 9, 3, 6, 7, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [9, 0, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [9, 4, 0, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [9, 4, 5, 3, 6, 0, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
[[3, 7, 1, 5, 4, 2, 8, 9, 6], [9, 4, 5, 3, 6, 7, 0, 1, 2], [0, 0, 0, 0, 9, 1, 7, 5, 0]
```

Figure 3: Working of Backtracking

I can't show the whole algorithm working because of the space issue but it will clear the concept of Backtracking. Firstly this shows the first 3 boxes how the values are inserting and then it keeps on repeating till the right value is found
Like if you see in the 2nd Box it starts with [0,0,0,3,6,0,0,1,2] after the 7th iteration it changes like [4,8,0,3,6,0,0,1,2] and it starts propagating for others then after the 5 more iterations it finds the value not good for the solution so it backtracks and box changes to [8,4,0,3,6,0,0,1,2] so this is how it works.

5 Relaxation Labeling on Sudoku

Relaxation Labeling is also found to be an active area of research this problem is still an NP-complete problem and yet many researchers are still trying to optimize the solution to ta relaxation labeling problem. The algorithm is a complex one it works on contextual information that is used for the labeling problems sometimes local info is not sufficient to bring a problem to a solution. The contextual information is expressed in a real-valued matrix $n^2 \times m^2$ of compatibility coefficient R defined as:

$$R = r_{ij}(\lambda, \mu) \tag{8}$$

where the coefficient measures the strength of compatibility of two hypotheses "b_i is labeled λ" and "b_j is labeled μ" initially the algorithm starts taking local measurements with assumption to provide each object $b_i \in$ B m-dimensional probability vector:

$$P_i^{(0)}\lambda = (P_i^{(0)}(1), ..., P_i^{(0)}(m))^T \tag{9}$$

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

so from the above points, we can conclude that the Sudoku is not a constraints satisfaction problem but also a contextual labeling problem. So basically this function in equation (9) is a probability distribution function that makes sure that the object I has a right label with λ and the concatenation of these probabilities defines a weighted labeling assignment $P^{(0)} \in \mathbb{R}^{nm}$ and the space of assignments defined as:

$$IK = \Delta^m = \Delta \times ... \times \Delta \tag{10}$$

where IK represents the unambiguous labeling assignment that is assigning the right label exactly to one object so we say the optimal solution will be the one that has no ambiguity, with the time the algorithm iterates thus resolving the ambiguity starting from the first labeling assignment that would be weighted and also not accepting the initial labels between the object and labels considering R_{ij} compatibility
According to the Heuristic formula given by Rosenfeld, Hummel, and Zucker state that in each rounds vectors probability updates with this ratio:

$$p_i^{(t+1)}(\lambda) = \frac{p_i^{(t)}(\lambda)q_i^{(t)}\lambda}{\sum_\mu p_i^{(t)}(\mu)q_i^{(t)}(\mu)} \tag{11}$$

Where

$$q_i^{(t)}(\lambda) = \sum_j \sum_\mu r_{ij}(\lambda, \mu)p_j^{(t)}(\mu) \tag{12}$$

it quantifies that the context gives at a time t to the hypothesis b_i is labeled with μ.

Now we need to consider the stopping criteria of the algorithm which in other words is so-called Average Local consistency and is given as :

$$A(p) = \sum_{i} \sum_{\mu} p_i(\mu) q_i(\mu) \tag{13}$$

this equation is a basic step of convergence that set the stopping criteria of an algorithm.

5.1 Coding and Implementation

There are many approaches to design the Relaxation labeling algorithm one with the matrices notation and the other with the formulas as aforementioned and we use only the implementing formulas but the process is slow in order to speed up the process we have to use both

```python
def RelaxationLabelingOnSudoku():
    global rij, p #rij coefficient
    diff = 1
    avg = 0
    t = 0
    rij = np.loadtxt("rij.csv", delimiter=",")  # creating Rij matrix on a sperate CSV file


    while diff > 0.001: #stopping criteria
        q = np.dot(rij, p) #dot product of coefficient and probability
        num = p * q
        row_sums = num.reshape(numberofcells*numberofcells, numberofcells).sum(axis=1)
        p = (num.reshape(numberofcells*numberofcells,numberofcells)/row_sums[:, np.newaxis]).reshape(729,1)
        avg = averageConsistency(q)
        diff = avg - avg_b
        avg_b = avg
        t += 1
        #above three line are actually related to convergence
    p = p.reshape(totalnumbercells, numberofcells)
```

According to the equation coated in (9) the probability distribution functions show the probability vector on objects and if it goes to the ambiguity it will then updates

```python

#here we will show compatibility matrix of a Sudoku
def CompatibilityMatrix(i, j, l, m):
    if i == j:
        return 0
    if l != m:
        return 1
    if SameRow(i,j) or SameColumn(i,j) or SameBlock(i,j): #checking the constraints
        return 0
    return 1
# it will only return 0 or 1 to show if assignment is valid or not

#according to formula we compute average consistency
def FindingAverageConsistency(q):
    return np.sum(p*q)


def solve_relaxationLabeling(SudokuBoard, create = False):
    global p, rij
    if create: createRijMatrix() # create matrix Rij when it is
    necessary
    VectorInitialization(SudokuBoard)

    rij = np.loadtxt("rij.csv", delimiter=",") # createRij()
    RelaxationLabelingOnSudoku()

    for i in range(len(SudokuBoard)):
        if i % 3 == 0 and i != 0:
            print("----------------------")

        for j in range(len(SudokuBoard[0])):
            if j % 3 == 0 and j != 0:
                print(" | ", end="")
            if j == 8:
                print(SudokuBoard[i][j])
            else:
                print(str(SudokuBoard[i][j]) + " ", end="")
```

This algorithm of labeling is very useful and handy but still, it has some issues regarding the probabilities if by chance 2 numbers have the same probability it will get confused and pick a number for 2 cells.

6 Comparison between Constraints Propagation Backtracking and Relaxation Labeling

I have tried many Sudoku Puzzles to run over both algorithms many times and notice a surprising change in Relaxation Labeling which behaves quite change in some puzzle like if I give an easy puzzle it jumps to a conclusion very quickly but when it faced a complex puzzle it gets confused and give the wrong solution. Each of the strategies has a different environment to tackle the puzzle each have their constraints and own properties in a different way but the goal is to find an optimized solution.
There are many ways we can do a comparison between these two algorithms but the best and unique comparison is of **Time** , **Space**, **Completeness** and **Optimality** these aspect best describes the comparison between them.

1. So in Time Complexity the **Constraint Propagation and Backtracking** have $\mathcal{O}(m^n)$ because this algorithm has to transverse the whole tree exploring all the states whereas in **Relaxation Labeling** the time depends on the updating cost which is $\mathcal{O}(n^2 \times m^2)$ so basically is term is unstable as it keeps on running until it reaches the stopping limit
2. As for Space is concerned the **Constraint Propagation and Backtracking** algorithm takes $\mathcal{O}(n \times m)$ which is linear as for the **Relaxation Labeling** as aforementioned we have two approaches to find the solution and both have their spatial complexity as for my approach it is compatibility matrix which takes also $\mathcal{O}(n \times m)$ space
3. **Constraint Propagation and Backtracking** is a complete algorithm in all the cases and it always finds a solution but as far as **Relaxation Labeling** is concerned it is not complete because it only works in few cases but not all the puzzles you can say complex ones.
4. If we talk about the optimality we can't say much about both the algorithms as in **Constraint Propagation and Backtracking** it finds all the possible solution so no optimal solution and for the **Relaxation Labeling** it can only solve the puzzle with very little complexity.

7 Conclusion

In a nutshell everything we have seen the two Algorithms so-called **Constraint Propagation and Backtracking** and **Relaxation Labeling** behave differently in different situations. This assignment is actually about the Constraints Satisfaction Problem and for this problem to solve we use two algorithms as mentioned above first we solve the Sudoku puzzle using Constraints propagation and then we backtrack to provide a solution close to optimal and then we find that the Sudoku is also a problem of Labeling and for this, we iteratively apply the heuristic function to update the states till the stopping limit is tackled We can also see that in this type of problem we cant use the brute force technique as this technique leads to approx $\times 10^{40}$ states that can only be performed by the supercomputer as far as the constraint propagation and backtracking is concerned it is a complete algorithm that takes time to solve the complex puzzle but will find a solution but for the Relaxation Labeling behaves very good in the less complicated puzzle but as we increase the difficulty it cant handle the puzzle or makes a right decisions