# Sudoku Solver using Constraint Propagation & Backtracking and Relaxation Labeling

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics


Artificial Intelligence: Knowledge Representation and Planning [CM0472-1]
Academic Year 2021 - 2022

**Student** Zuliani Riccardo 875532

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Sudoku Game

The Sudoku, in Japanese 数独, complete name 数字は独身に限る **Sū**ji wa **doku**shin ni kagiru, which in English mean "only solitary numbers are allowed" [5], is a logic game where a grid $9 \times 9$ is given to the player. The goal is to fill each cell with a number in the range from 1 to 9 such that each row, each column and each $3 \times 3$ box have all the range number without repetition. Initially the given board has some cell already filled with a number and this helps the player to deduce the possible choice that he/she can do in a free cell to complete the all board using the mentioned rules.

An example of initial board and solved board is shown in the two figure bellow:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 1.1: Initial Sudoku Grid

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 1.2: Solved Sudoku Grid

The starting Sudoku board affect the possibility to have only one configuration solution or multiple configurations. It has been proved that if the initial Sudoku board present at least 8 of the 9 possible number there is a unique solution, otherwise if there are only 7 number the solutions are at least two[1]. Moreover the initial grid affects also the strategy to find a reasonable solution. One of the most common is the *Brute Force* approach in which we generate all the possible solution by filling all the cells with all the possible range value and then analyse all the generated configurations and take only the one that satisfies the rules. Of course this technique has a major disadvantages which is the space complexity, therefore it is not an advisable technique.

A better strategy is to take benefits by the present fixed cells which suggest us the possible values that free cell could take satisfying the rules.

# Chapter 2

# Sudoku as Constraint Satisfaction Problem

The Sudoku game can be viewed as a **Constraint Satisfaction Problem**, in this type of problem each state is seen as a set of variables each of which has a value. The problem is solved when each variable has a value that satisfies all the constraints on the variable, for constraints in other words we consider the problem rules.

In mathematic language the Constraint Satisfaction problem is defined by three main components *X, D* and *C* like so [3]:

- A set of variable $X = \{x_1, ..., x_n\}$
- Each variable $x_i$ has its associated domain $D = \{d_1, ..., d_n\}$ which is composed by all possible value that the specific variable can take
- A set of constraints $C$ that specifies the rules for each domain variable

In the Sudoku word we can translate these three main concept in a more understandable way:

- The set of variable $X$ is equal to the set of all 81 cells
- Each cell has a domain $D$ which is the set of number from 1 to 9
- $C$ is the set of rule that define the domain of each cell. Here we have two type of constraint:
  - **Direct Constraint:**
    - *Row Constraint*, in which each value in the same row have to be different
    - *Column Constraint*, in which each value in the same column have to be different
    - *Box Constraint*, in which each value in the same $3 \times 3$ box have to be different
  - **Indirect Constraint:** say that each range number have to appear in each row, column and box. This constraint take into account the already filled cells to restrict the domain non non free cell

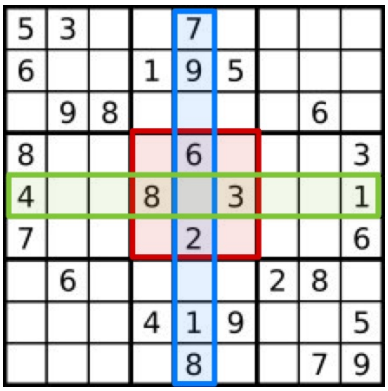


Figure 2.1: Direct Constraint

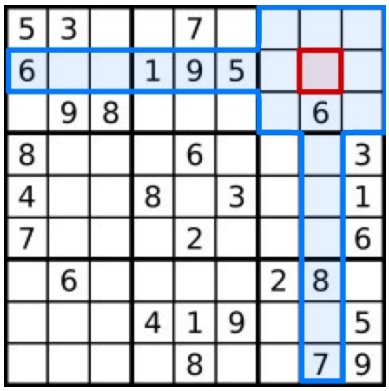


Figure 2.2: Indirect Constraint

In figure 2.1 as we can see the red box is referred to the *Box Constraint*, the blue vertical row is referred to the *Row Constraint* and green horizontal column is referred to the *Column Constraint*.

On the other hand in the second figure 2.2 we are taking into account the cell $X_{2,8}$ and its domain which initially is formed by $D_{2,8} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Now we have to update $D_{2,8}$ by removing the value of the corresponding cell value of the same row $\{6, 1, 9, 5\}$, of the same column $\{6, 8, 7\}$ and of the same box $\{6\}$. With so we end up with all the possible value that the cell $X_{2,8}$ can take during the computation of the solution, which in this example are $\{2, 3, 4\}$.

# Chapter 3

# Constraint Propagation & Backtracking

In this chapter we describe and analyse the Constraint Propagation & Back Tracking technique used to solve the Sudoku puzzle.

## 3.1 Description

So as mentioned at the start of the report try all possible solution and take only the solution that satisfy the constraint is not a highly recommended. Instead we can adopt the **Constraint Propagation** method. The main effect that this strategy carries with it is that it choose the possible value to fill a specific cell in depends on its actual domain set, which in turn is dependent on the constraint rule that we have seen in the previous chapter. So starting from this when a free cell is filled with one of its domain value the algorithm propagate the new update puzzle domain by removing the specific value from the domain of free cells that not satisfy the constraint rules.

This type of strategy is very powerful for easy initial configuration, but when it approaches some high difficult initial grid, it is possible that the made choice could lead to a dead end. So we need also a strategy to get **back** into a previous strategy restoring the filled cell into free cell and try an other value, which could take us to a satisfying solution or maybe not.

This is the main idea of the other strategy called **Backtracking**. It is used for problem that generate a large state space tree, in which each node represent a a possible state of the problem. The **Backtracking** algorithm use a *depth-first search* for choosing values for one variable at a time and backtracks when a variable has no legal values left to assign. In a brief the algorithm choose an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then the recursion process return a failure, causing the previous step to try an other value [3].

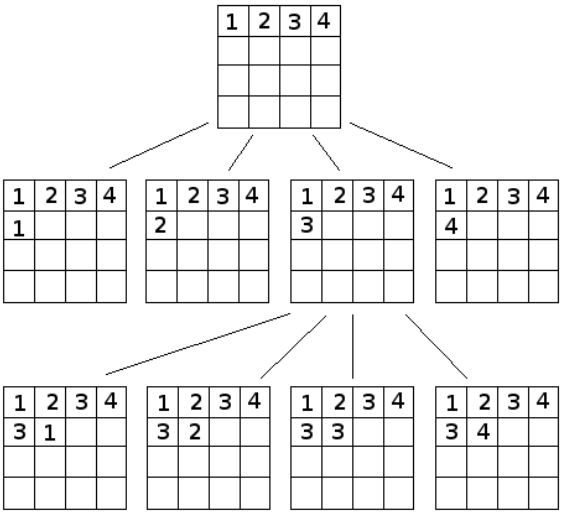


Figure 3.1: Backtracking in a Sudoku $2 \times 2$

One of the main problem of Backtracking is that the continuous recursive process could take into a count an high amount of memory for big grid since it basically explores all the possible states.

So a good solution is to combine these two strategy. **Constraint Propagation** reduce the searching space by trying only the satisfying domain value for each cell and in case of dead point there is the **Backtracking** approach that permit to get back to the previous state to try an other domain value.

## 3.2 Software implementation

In this section we discuss the general behaviour, software implementation and adopted choice to solve a Sudoku puzzle using the Constraint Propagation and Backtracking technique.

```
def solveCP(bo, exp, back):
    domain = getDomainCells(bo)
    # Get all the cells domain

    location = getNextMinimumDomain(domain, bo)
    # Get the location of the next cell with minimum domain length

    if(location is None): return (True, bo, exp, back)
    row, col = location

    for val in domain[row * DIMENSION + col]:
        if checkValidAssign(copy.deepcopy(bo), row, col, val):
            # If the assignement is valid we update the board
            bo[row][col] = val

            # Recursion call
            solved, board, expanded, backword = solveCP(
                bo, exp+1, back
            )

            # If a valid solution is found we increase the stats
            if(solved):
                return (True, board, exp + expanded, back + backword)

        # Otherwise we get back to the previous step and check
        # another value
        back += 1
        bo[row][col] = 0

    # If we have tried all possible cell domains no solution is found
    return (False, bo, exp, back)
```

The function *solveCP* is the main function to solve the Sudoku using the Constraint Propagation and Backtracking technique and we start the discussion from this. Initially in each recursive call we get our the domain of our grid by the function *getDomainCells(bo)* in this function we apply the constraints to return a list of the corresponding domain of each cell at that moment.

Moreover we get into *getNextMinimumDomain(domain, bo)*:

```
# Function to get the position of the cell with the minimum length domain
def getNextMinimumDomain(domain, bo):
    listMapDomain = getListMapDoamin(domain, bo)
    minimumFirstList = min(listMapDomain)
    if minimumFirstList == EMPTYDOMAINVALUE: return None
    index = listMapDomain.index(minimumFirstList)
    return(int(index / DIMENSION), index % DIMENSION)
```

This function is used to get the location of the next cell having the minimum length domain. We adopted this strategy because trying to guess the correct domain of a cell having the smallest domain length bring with it an higher probability to make the correct choice rather than choose an higher domain length cell. For example: let $A$ and $B$ two cell with respective domain $D_A = \{1, 2, 3, 4, 5\}$ and $D_B = \{5, 4, 3\}$. The *getNextMinimumDomain* will choose first $D_B$ because has a smaller length than $D_A$, since the probability to guess the correct cell value of $B$ is equal to $\frac{1}{3} = 0.33\overline{3}$ which is more than the probability to guess the correct cell value of $A$ whose is equal to $\frac{1}{5} = 0.2$.

Continuing if *getNextMinimumDomain* return a *None* location this means that the cell with minimum domain length has length equal to 10, which is impossible because the range of possible value is from 1 to 9 so the length is 9. We use this trick to avoid the repicking of a cell that has already filled with a satisfying constraints value. So if the minimum length is equal to 10 also all the other domains of the other cells will have length equal to 10 and this means that we have found a correct solution.

Furthermore if the location variable is really a location we extract the respective row and column and we loop for each value of the calculated cell domain. In this loop we verify if every domain value is a valid assign to the cell, if so we update the board and make a recursion call. If the computation is finished we check if the puzzle is solved and, in case, return the updated statistics.

Otherwise if we the assignment is not valid we restore the cell value to a free cell and try an other value. If the calculation ends with all tested cells domains, no solution has been found.

# Chapter 4

# Relaxation Labeling

In this chapter we describe and analyse the Relaxation Labeling technique used to solve the Sudoku puzzle.

## 4.1 Description

An other interesting technique is the so called Relaxation Labeling which consist of not use only local information but also consider contextual information. Relaxation labeling processes are a class of mechanisms that were originally developed to deal with ambiguity and noise in vision systems. The general framework, however, has far broader potential applications and implications. The structure of relaxation labeling is motivated by two basic concerns [2]:

1. The decomposition of a complex computation into a network of simple "myopic," or local, computations
2. The requisite use of context in resolving ambiguities

A general labeling problem is composed by:

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

But we are focusing to solve a specific problem not a general one.The Sudoku world viewed as a labeling problem has:

- A set of $n = 81$ cells $B = \{b_1, ..., b_n\}$
- A set of $m = 9$ labels $\Lambda = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, that represent the possible value of a cell

The main goal is to assign a label of $\Lambda$ to each object of set $B$. The contextual information is represented as a matrix of compatibility coefficients:

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

This is an important statement of the technique, because it measure the strength of compatibility between two hypothesis: "$b_i$ is labeled with $\lambda$" and "$b_j$ is labeled with $\mu$, where $b_i$ and $b_j$ are two different object form the set of objects $B$.

In our case the compatibility matrix is defined to return 1 if the assignment is valid, and 0 otherwise. And it is defined like so:

```python
# Function to define the compatibility between cell 'i' and
# label 'lb' and cell 'j' and label 'mu'
def compatibility(i, j, lb, mu):
    if i == j: return 0
    if lb != mu: return 1
    if sameRow(i, j) or sameCol(i, j) or sameBox(i, j): return 0
    return 1

def sameRow(i, j): return i[0] == j[0]
def sameCol(i, j): return i[1] == j[1]
def sameBox(i, j): return ((i[1] // 3) == (j[1] // 3) and
                           (i[0] // 3) == (j[0] // 3))
```

Initially every relaxation labeling problems starts with an initial m-dimensional probability vector for each object, representing the probabilities that an object can take a specific label.

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

With m = 9, $p_i^{(0)}(\lambda) \geq 0$ and $\sum_\lambda p_i^{(0)}(\lambda) = 1$. Where $p_i^{(0)}(\lambda)$ is the probability distribution at time 0, that object $i$ is labeled with the label $\lambda$. For example if a given cell has the respective possible values domain equal to [1,2,4] then $p_i^{(0)}(\lambda) = (0.33\overline{3}, 0.33\overline{3}, 0, 0.33\overline{3}, 0, 0, 0, 0, 0)$.

Each object has a probability distribution, and by union of these vector we define a weight labelling assignment $p^{(0)} \in \mathbb{R}^{mn}$. All the possible weight labelling assignment belongs to a space IK defined like so:

$$IK = \Delta^m = \Delta \times ... \times \Delta$$

Where each $\Delta$ is an alias of $\mathbb{R}^n$.

Each vertex of IK represent an **Unambiguous Labeling Assignment**. In other word we have the certainty that only a single label could be assign to a specific object, therefore the optimal solution and case consist in being in this situation in every object, since there is not ambiguity. The idea of reducing the possible ambiguity is made possible by considering the compatibility $r_{i,j}$ previous defined between objects and labels. The problem is that it is not always possible at the end of the iterative procedure to have all probability distributions with a single label more probable than the others. Thus the Relaxation Labeling technique **does not guarantee convergence,** and therefore it is possible that at the end of the iteration we obtain an unsatisfactory solution, consisting in unambiguous assignment of label that do not satisfy the constraints rule. Now we analyse an example of vectors that provide ambiguity or inconsistent assignments:

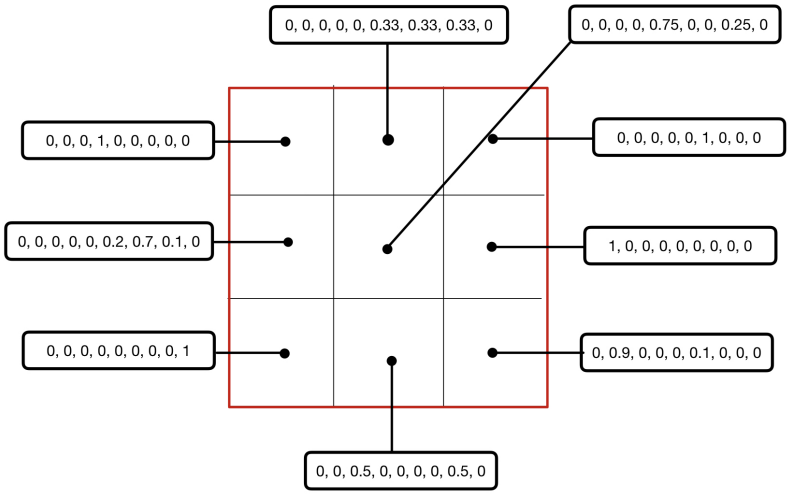


Figure 4.1: Example of possible weight labeling assignment

As we can see form the image 4.1 we can distinguish three type of assignments:

- **No uncertain assignments:** for cells $C_{1,1}$, $C_{3,1}$, $C_{1,3}$ and $C_{2,3}$. Because we are in the optimal situation where a single label has probability 1 so we are sure that is the best and only choice that we can take
- **Quite uncertain assignments:** for cells $C_{2,1}$, $C_{3,3}$ and $C_{2,2}$. Because there is a label that is most likely to be the best choice for that particular cell.
- **Uncertain assignments:** for cells $C_{3,2}$ and $C_{1,2}$. And here we have a total uncertainty regarding what label to choose because the once that have probability different from zero have the same chance to be assign

Now we face to the strategy on how Relaxation Labeling tries to update the probability vectors to obtain the best suitable solution. In each iteration the vector of probability distribution are updated using the following heuristic formula provided by Rosenfeld, hummel and Zucker in 1976:

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

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

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

Moreover, we have to introduce a stopping criteria to decide when to stop the computation. A classic strategy is to use the so called **Average Local Consistency:**

$$A(p) = \sum_i \sum_\lambda p_i(\lambda) q_i)(\lambda)$$

*Marcello Pelillo*, in 1997, prove that the algorithm strictly increases the average local consistency on each iteration such that:

$$A(p^{t+1}) > A(p^t)$$

For $t$ = 0,1,2,... until a fixed point is reached. Using this theorem we can define a simple stopping criteria like the difference between the average local consistency of time t+1 and the previous one at time t, and we loop until this difference is smaller than a given threshold $\epsilon \in \mathbb{R}$.

$$A(p^{t+1}) - A(p^t) > \epsilon$$

An other strategy is to use the **Euclidean Distance** between the probability vectors of time $t + 1$ and time $t$. We decide to use this strategy since our main aim is to maximize the objective function and we want to take into account not only the output but also the input.

$$\sqrt{\sum_{k=1}^{n}(p_k - q_k)^2} > \epsilon$$

Where $P = (p_1, ..., p_n)$ represent the probability vector at the current step and $Q = (q_1, ..., q_n)$ represent the probability vector at the previous step [4]. We iterate until the Euclidean Distance between this two vectors is less than a given threshold $\epsilon \in \mathbb{R}$ which in our experiment is set to 0.001.

## 4.2 Software implementation

```
# Main function to solve the sudoku board using Relaxation Labelling
def solveRL(matrix):
    domain = getDomainCells(matrix)
    # Get all the cells domain

    probDist = defineDistributions(domain)
    # Get the vector of probability distribution

    diff = 1
    iterations = 0
    oldProb = copy.deepcopy(probDist)

    while diff > 0.001:
    # 0.001 is the threshold for the stopping criteria

        computeAllQ(probDist)
        # Compute all hypothesis "b_i is labeled with lambda"

        computeAllP(probDist)
        # Compute all new probability distributions

        diff = euclideanDistance(probDist, oldProb)
        # Compute the Euclidean distance

        oldProb = copy.deepcopy(probDist)
        iterations += 1

    # Return the best fittable value for each cell of the grid
    return chooseBestFittableValue(matrix, probDist), iterations
```

The first operation like in Constraint Propagation & Backtracking is to get the initial domain list of all cells, then we can obtain the associated domain probability distributions, and loop until we reach convergence. During this loop we continuous update the probability distribution as well as the compatibility matrix.

# Chapter 5

# Benchmarks and Results

The benchmarks are done taking into account 5 Sudoku configuration of 4 different difficulty, respectively easy, normal, medium and hard.

## 5.1 Constraint Propagation & Backtracking

| Filename | Execution Time | Expanded Nodes | Backward Nodes | Solved |
|---|---|---|---|---|
| easy1.txt | 0.035142 | 1081 | 0 | True |
| easy2.txt | 0.030901 | 861 | 0 | True |
| easy3.txt | 0.024801 | 990 | 0 | True |
| easy4.txt | 0.016337 | 820 | 0 | True |
| easy5.txt | 0.021922 | 946 | 0 | True |
| normal1.txt | 0.0278 | 1378 | 0 | True |
| normal2.txt | 0.025671 | 1326 | 0 | True |
| normal3.txt | 0.027276 | 1225 | 34 | True |
| normal4.txt | 0.024956 | 1225 | 25 | True |
| normal5.txt | 0.025779 | 1326 | 0 | True |
| medium1.txt | 0.073757 | 1378 | 164 | True |
| medium2.txt | 0.031849 | 1596 | 32 | True |
| medium3.txt | 0.080914 | 1596 | 204 | True |
| medium4.txt | 0.058402 | 1431 | 45 | True |
| medium5.txt | 0.0968 | 1596 | 123 | True |
| hard1.txt | 0.080047 | 1711 | 125 | True |
| hard2.txt | 0.089135 | 1596 | 174 | True |
| hard3.txt | 0.033165 | 1711 | 21 | True |
| hard4.txt | 0.033071 | 1770 | 0 | True |
| hard5.txt | 0.033619 | 1830 | 0 | True |

Table 5.1: Results of Constraint Propagation and Backtracking

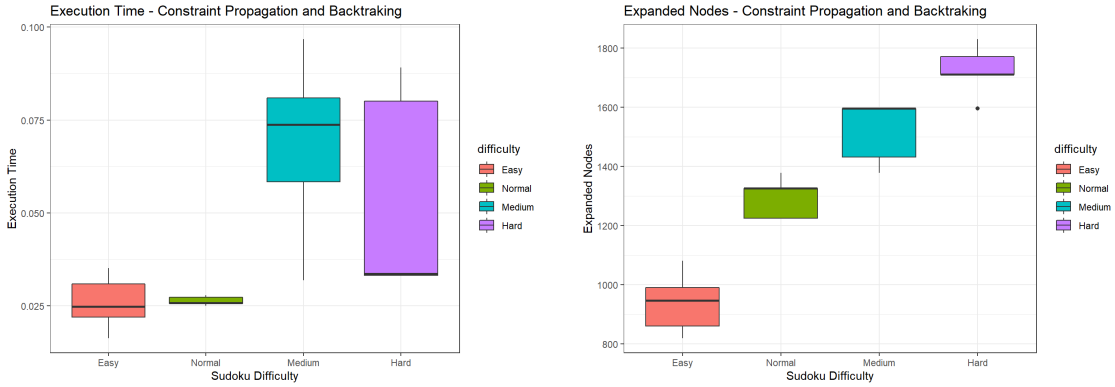


Figure 5.1: Execution Time and Expanded Nodes for Constraint Propagation & Backtracking

As we can see from the table 5.1 and the second boxplot 5.1, all the board was solved with a directly proportional execution time per difficulty. But we notice from the first boxplot that the medium sudoku on average takes more time than the hard one, but on the other hand the hard sudoku are solved with an higher number of expanded node. All of this happened because the medium sudokus have on average a number of backword nodes higher than the hard sudokus.

## 5.2 Relaxation Labeling

| Filename | Execution Time | Number of Iteration | Solved |
|---|---|---|---|
| easy1.txt | 16.687074 | 162 | True |
| easy2.txt | 17.864489 | 170 | True |
| easy3.txt | 15.429264 | 141 | True |
| easy4.txt | 13.692886 | 141 | True |
| easy5.txt | 9.930051 | 105 | True |
| normal1.txt | 46.143734 | 493 | False |
| normal2.txt | 21.522013 | 232 | False |
| normal3.txt | 22.042598 | 237 | False |
| normal4.txt | 24.301476 | 262 | False |
| normal5.txt | 36.064446 | 387 | False |
| medium1.txt | 30.059299 | 324 | False |
| medium2.txt | 39.82368 | 422 | False |
| medium3.txt | 37.302867 | 398 | False |
| medium4.txt | 25.848797 | 276 | False |
| medium5.txt | 21.293147 | 229 | False |
| hard1.txt | 29.594862 | 319 | False |
| hard2.txt | 32.657705 | 353 | False |
| hard3.txt | 41.004882 | 444 | False |
| hard4.txt | 28.971116 | 312 | False |
| hard5.txt | 42.272137 | 456 | False |

Table 5.2: Results of Relaxation Labeling

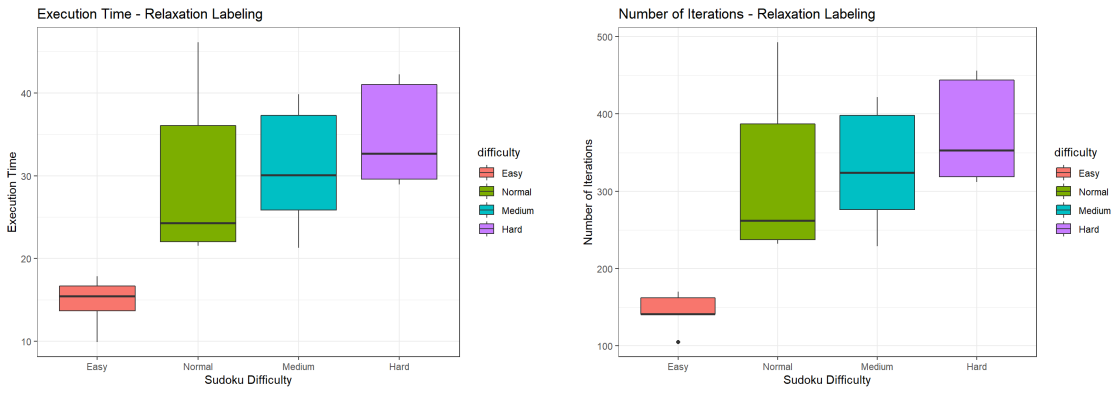


Figure 5.2: Execution Time and Number of Iteration for Relaxation Labeling

On the other hand regarding the Relaxation Labeling the first things that comes to our eyes watching the table 5.2 is that only the group of easy Sudoku was solved correctly and all the rest was not resolved even if the convergence point have been reached. Moreover as expected the Execution Time and the Number of Iteration are directly proportional at the difficulty level.

## 5.3 Technical Analysis

In this section we look into the technical algorithmic aspects of the two use strategy. By first we analyse the **Constraint Propagation and Backtracking** strategy:

- *Completeness:* reached, because it finds a satisfying constraints solution
- *Optimality:* reached, if we assume that a solution exists, the strategy always find it using the minimum necessary steps
- *Time Complexity:* $O(m^n)$
- *Space Complexity:* $O(n)$ for the domain matrix and $O(n \times m)$ for the domain matrix
- *Stopping Criteria:* when it finds a satisfying constraints solution or in the other case when it loops in all the search space without finding a solution

And now we analyse the **Relaxation Labeling** strategy:

- *Completeness:* not reached, because it solves only the easy Sudoku
- *Optimality:* not reached, because it solves only the easy Sudoku
- *Time Complexity:* $O(t \times m^5)$ which is the complexity of the heaviest computation function, *computeAllQ*, and $t$ is the unknown number of loops done by the while
- *Space Complexity:* $O(m \times n)$ for the probability distribution vectors and the compatibility matrix
- *Stopping Criteria:* use an heuristic like the Average Local Consistency or the used Euclidean Distance

At all points $n$ equals the number of cells, so in our case 81, and $m$ equals the domain length of the cell, which in our case is 9.

# Chapter 6

# Conclusion

In this little report we analysed two common strategy to solve the Sudoku puzzle as a Constraint Satisfaction Problem. The first one based on Constraint Propagation and Backtracking technique to provide completeness propriety as shown in the previous page. And the second one based on the idea to cast our problem into a labeling problem to take advantage not only from the local information but also considering the contextual information.

As shown from the table analysis the first method is able to find a satisfying solution, if it exists, but the complexity in term of time and expanded nodes increase drastically with the complexity of the initial puzzle configuration.

On the other hand the Relaxation Labeling strategy can solve only Sudoku with an easy initial configuration, otherwise if the difficulty increase it is not able to take the right value choice for each cell, resulting in an unsatisfying solution.

In conclusion, since the completeness is the first propriety to be taken into account and one of the most important in evaluating an algorithm, we end up saying that the Constraint Propagation and Backtracking technique is preferable to the Relaxation Labeling. However, remembering that its only weakness is related to the complexity of the initial configuration matrix.

# Bibliography


[1] Agnes Herzberg and Ram Murty. "Sudoku squares and chromatic polynomials". In: *Internationale Mathematische Nachrichten* 54 (June 2007).

[2] Robert A. Hummel and Steven W. Zucker. "On the Foundations of Relaxation Labeling Processes". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-5.3 (1983), pp. 267–287. DOI: 10.1109/TPAMI.1983.4767390.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. 3rd ed. Pearson, 2009.

[4] Wikipedia. *Euclidean distance - Wikipedia*. URL: https://it.wikipedia.org/wiki/Distanza_euclidea.

[5] Wikipedia. *Sudoku - Wikipedia*. URL: https://it.wikipedia.org/wiki/Sudoku.