
Università
Ca'Foscari
Venezia

# *ARTIFICIAL INTELLIGENCE: KNOWLEDGE REPRESENTATION AND PLANNING*

## *CM 0472-1*

## ASSIGNMENT 1

**Student:**

Giosuè Zannini 873810

Academic year 2021/2022

# Contents

**1 Introduction** **1**

1.1 Problem . . . . . 1

1.2 What's Sudoku . . . . . 1

**2 Approach with Constraint Propagation and Backtrackingrute** **2**

2.1 My first idea of implementation . . . . . 3

2.2 My second idea of implementation . . . . . 5

**3 Approach with Relaxation Labeling** **7**

3.1 My first idea of implementation . . . . . 9

3.2 My second idea of implementation . . . . . 10

**4 Conclusions** **11**

4.1 Performance for each approaches . . . . . 11

4.2 Comparison of all approaches . . . . . 15

# Chapter 1

# Introduction

## 1.1 Problem

In this assignment I have to write a solver for Sudoku puzzles using a Constraint Satisfaction approach based on Constraint Propagation and Backtracking, and one based on Relaxation Labeling. Moreover I have to compare the approaches, their strengths and weaknesses.

## 1.2 What's Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle.
In classic Sudoku, the objective is to fill a 9 × 9 grid with digits so that each column, each row, and each of the nine 3 × 3 sub-grids that compose the grid (also called "boxes", "blocks", or "regions") contain all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution. French newspapers featured variations of the Sudoku puzzles in the 19Th century, and the puzzle has appeared since 1979 in puzzle books under the name Number Place. However, the modern Sudoku only began to gain widespread popularity in 1986 when it was published by the Japanese puzzle company Nikoli under the name Sudoku, meaning "single number".

# Chapter 2

# Approach with Constraint Propagation and Backtrackingrute

This technique is based on two different techniques:

- Backtracking: is a general algorithm for finding solutions to some computational problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution. Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. Backtracking depends on user-given "black box procedures" that define the problem to be solved, the nature of the partial candidates.

- Constraint Propagation: is the process of finding a solution to a set of constraints that impose conditions that the variables must satisfy. A solution is therefore a set of values for the variables that satisfies all constraints—that is, a point in the feasible region. The techniques used in constraint satisfaction depend on the kind of constraints being considered. Often used are constraints on a finite domain, to the point that constraint satisfaction problems are typically identified with problems based on constraints on a finite domain.

The merge of these techniques generate the method that I use as first approach. With this I can use a Backtracking in better way, indeed it abandons a candidate as soon as it determines that the candidate cannot possibly be completed to a valid solution.

## 2.1 My first idea of implementation

As first implementation I decided to scroll through the whole array row by row and for each blank cell I created a "domain"($\mathbb{D}$) in which there are all possible numbers that can be written in the cell. To determine the domain ($\mathbb{D}$) for each blank cell $i$ I must control each row, column, and square control cell related to $i$, and creating a set($\mathbb{I}_i$) with the numbers present in row, column, and square. Using this information I determine the set($\mathbb{D}_i$) of numbers that can be used for each cell $i$.
This is the formula that I use to determine the domain for each blank cell $i$:

$$\begin{aligned} &\mathbb{C} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ &\mathbb{D}_i = \mathbb{C} \setminus \mathbb{I}_i \end{aligned} \tag{2.1}$$

The above step is used within the Sudoku resolution algorithm. Being that I will go to visit a tree, I preferred to implement a recursive procedure as this type of structure lends itself well to this kind of visit. The algorithm starts by taking the first empty cell determined by the matrix scan, generates the associated domain and tries to enter a number in the Sudoku in the domain, then makes a recursive call to go deeper into the tree. If at some point an empty cell has a zero domain, the tree will be returned and a new branch will be tried. This is possible thanks to the stack that will keep in memory the previous domain and the choice made by it.

Below the pseudo code of the algorithm just described:

```
1: explore_solutions()
2: d ← take_the_next_blank_cell()                    ▷ Take the next white space
3: possible_number ← take_domain(d)                  ▷ Pointer to possible numbers
4: while possible_number != empty() && !sudoku_is_complete() do
5:     write_number_on_matrix()                      ▷ write number inside sudoku
6:     res ← explore_solutions()
7:     if (!res) then                                ▷ in this case rebuild the sudoku position
8:         write_blank_on_matrix()
9:     end if
10: end while
11: if (sudoku_is_complete()) then
12:     true
13: else
14:     false
15: end if
```

## 2.2 My second idea of implementation

This second approach is very similar to the previous one except for the choice of the empty cell to fill. Instead of using a static order for what concerns the choice, I opted for a dynamic order. the two main techniques we have seen are:

1. Most Constrained Variable: pick variable with fewest legal values to assign next (minimizes branching factor).

2. Least Constrained Variable: pick variable that rules out the fewest values from neighboring domains (maximizes branching factor).

In this report I used the 1 technique, so I decided to implement a map where the key is the location of the empty cell and the value is the domain of the same cell (same domain used for the previous technique), this to be able to access a location given coordinates ($O(1)$). I also used a heap structure which will keep all pointers to domains. This structure is useful because it is simple and performing to maintain a min heap sorted by the size of the set of numbers that each cell can use.

Also in this case I implemented a recursive algorithm, below the pseudo code of the main function:

```
1: explore_solutions()
2: d ← take_the_smallest_blank_cell()          ▷ Take the smallest white space
3: possible_number ← take_domain(d)            ▷ Pointer to possible numbers
4: while possible_number != empty() && !sudoku_is_complete() do
5:     write_number_on_matrix()                ▷ write number inside sudoku
6:     res ← explore_solutions()
7:     if (!res) then                          ▷ in this case rebuild the sudoku position
8:         write_blank_on_matrix()
9:     end if
10: end while
11: if (sudoku_is_complete()) then
12:     true
13: else
14:     false
15: end if
```

# Chapter 3

# Approach with Relaxation Labeling

Relaxation labelling techniques can be applied to many areas of computer vision. Relaxation techniques have been applied to many other areas of computation in general, particularly to the solution of simultaneous nonlinear equations. The basic elements of the relaxation labelling method are a set of features belonging to an object and a set of labels. In the context of vision, these features are usually points, edges and surfaces. Normally, the labelling schemes used are probabilistic in that for each feature, weights or probabilities are assigned to each label in the set giving an estimate of the likelihood that the particular label is the correct one for that feature. Probabilistic approaches are then used to maximise (or minimise) the probabilities by iterative adjustment, taking into account the probabilities associated with neighbouring features. Relaxation strategies do not necessarily guarantee convergence, and thus, I may not arrive at a final labelling solution with a unique label having probability one for each feature.

1. $O$ is the set $\{o_1, \ldots, o_n\}$ of $n$ object features to be labelled.

2. $L$ is the set $\{l_1, \ldots, l_m\}$ of $m$ possible labels for the features.

Let $\mathbb{P}_i(I_k)$ be the probability that the label $l_k$ is the correct label for object feature $o_i$. The labelling process starts with an initial, and perhaps arbitrary, assignment of probabilities for each label for each feature. The basic algorithm then transforms these probabilities into to a new set according to some relaxation

schedule. This process is repeated until the labelling method converges or stabilises. This occurs when little or no change occurs between successive sets of probability values. Here an operator considers the compatibility of label probabilities as constraints in the labelling algorithm. The compatibility $R_{ij}(l_k, l_q)$ is a correlation between labels defined as the conditional probability that feature $o_i$ has a label $l_k$ given that feature $o_j$ has a label $l_q$ i.e. $R_{ij}(l_k, l_q) = \mathbb{P}(l_k|l_q)$. Thus, updating the probabilities of labels is done by considering the probabilities of labels for neighbouring features. In a now classic 1976 paper, Rosenfeld, Hummel, and Zucker introduced heuristically the following update rule:

$$p_i^{(t+1)}(\lambda) = \frac{p_i^{(t)}(\lambda) q_i^{(t)}(\lambda)}{\sum_\mu p_i^{(t)}(\mu) q_i^{(t)(\mu)}} \tag{3.1}$$

Where

$$q_i^{(t)}(\lambda) = \sum_j \sum_\mu r_{ij}(\lambda, \mu) p_i^{(t)(\mu)} \tag{3.2}$$

quantifies the support that context gives at time $t$ to the hypothesis "$b_i$ is labeled with label $\lambda$". The average local consistency of labeling $p$ is defined as:

$$A(p) = \sum_i \sum_\lambda p_i(\lambda) q_i(\lambda) \tag{3.3}$$

## 3.1 My first idea of implementation

As first implementation I decided to build a map as a matrix where, the key are the position of the blank cell $i$ and the value is a probability vector where each place $n$ is the probability that $i$ holds the value $n$. To build this vector I still used the domain shown in section 2.1, and after determining the domain for each cell I assigned to each present value a probability in accordance with the uniform distribution. In this way I don't have to use all the Sudoku cells but only the empty cells. Therefore $O$ is set of blank cells and $L$ is set of value from 1 to 9. So I can obtain the $P$ matrix at time $0$ and with equation 3.2 can obtain the $Q$ matrix that holds all $o \in O$ and all $l \in L$. In this specific problem the compatibility function $R_{ij}(\lambda, \mu)$ has the following behaviour:

- $0$ if $i = j$ OR ($i$ and $j$ are related (so they are in the same row/column/square) AND $\lambda = \mu$)
- $1$ otherwise

Now the algorithm to solve Sudoku is quite simple, in fact it uses the equation 3.1 to build the matrix $P$ at time $t + 1$ and the equation 3.2 to build the matrix $Q$ at time $t + 1$. This cycle continues until the average consistency illustrated in equation 3.3 of $P^{(t)}$ is very close to the average consistency of $P^{(t+1)}$ (The limit threshold adopted for the stopping criteria is 0,01).

Below the pseudo code just illustrated:

```
1: compute_solution()
2: while dst < 0.01 do
3:     dst ← average_consistency()          ▷ Calculate the consistency of P^(t)
4:     compute_P(t+1)
5:     compute_Q(t+1)
6:     P^(t) ← P^(t+1)
7:     dst ← abs(dst - average_consistency())   ▷ Calculate the distance of
   consistency from P^(t+1) to P^(t)
8: end while
```

## 3.2 My second idea of implementation

This approach is practically the same as the overlying method, the only difference is that I tried to improve the performance with a bit of parallelization by using OpenMP directives to decrease the calculation time of the equations: 3.1, 3.2 and 3.3.

# Chapter 4

# Conclusions

## 4.1 Performance for each approaches

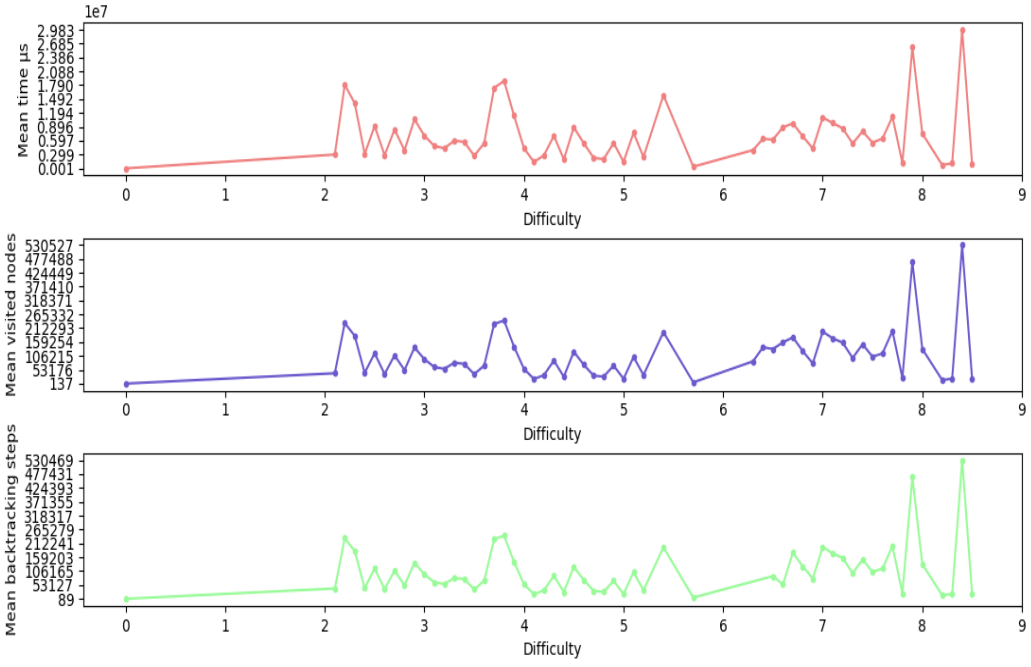
In this section, I go to view and analyze the results from the dataset Sudoku-3m.csv among all approaches that I tried. This dataset is characterized by 3 million Sudoku, each with a difficulty value that varies between $0.0$ and $8.5$, it is computed by an automated solver and it is based on the average search tree depth over 10 attempts. To edit the following tables I used the following hardware:

- Processor: AMD Ryzen 5 3500U
- Ram: 8GB DDR4
- Graphic card: AMD Radeon Vega 8

First I want to show for each approach a tiny summary for all techniques:

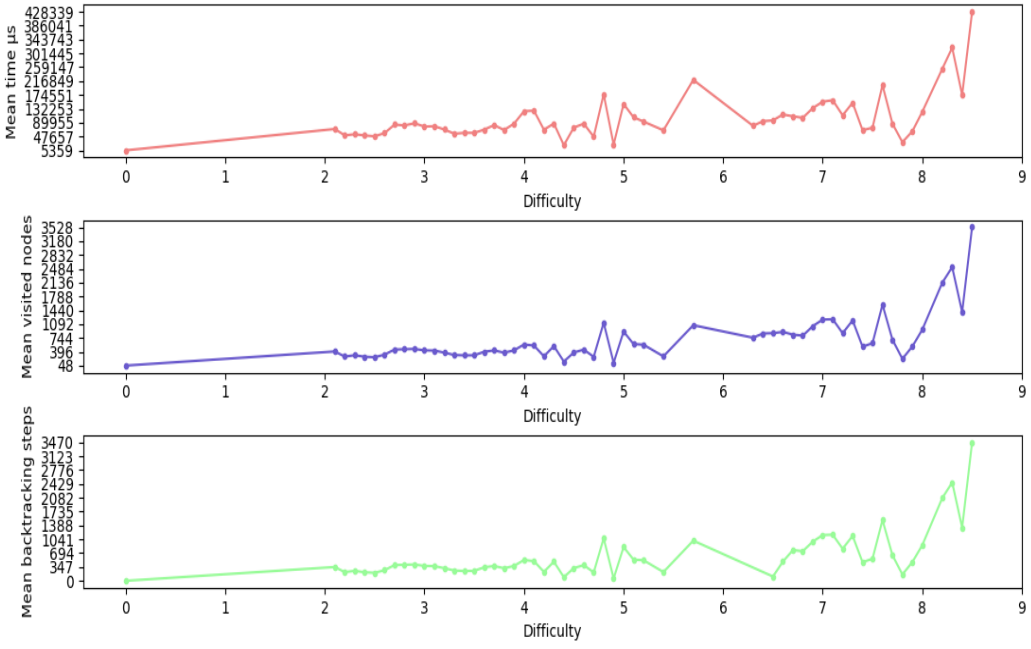
- First approach addressed in the section 2.1:

| Difficulty | Mean time ($\mu$s) | Mean visited nodes | Mean backtracking steps | Solved |
|---|---|---|---|---|
| 0,0 | 11674,55 | 138 | 89 | yes |
| 2,1 | 2981786,54 | 39634 | 39576 | yes |
| 3,0 | 7002440,04 | 93298 | 96927 | yes |
| 4,0 | 4719154,17 | 60637 | 55876 | yes |
| 5,0 | 1437041,50 | 18654 | 18596 | yes |
| 6,3 | 3941336,27 | 85272 | 55120 | yes |
| 7,0 | 10971318,12 | 198795 | 198737 | yes |
| 8,0 | 7501109,33 | 132573 | 132515 | yes |



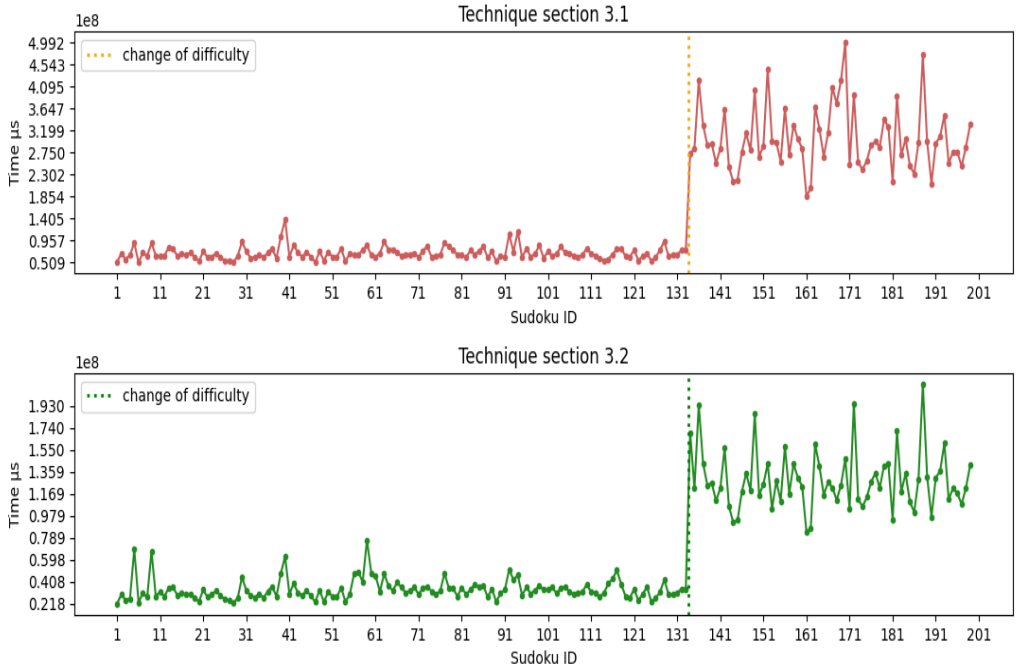
- Second approach addressed in the section 2.2:

| Difficulty | Mean time ($\mu$s) | Mean visited nodes | Mean backtracking steps | Solved |
|---|---|---|---|---|
| 0,0 | 5359,26 | 48 | 0 | yes |
| 2,1 | 70560,62 | 403 | 346 | yes |
| 3,0 | 77996,42 | 431 | 376 | yes |
| 4,0 | 124760,86 | 580 | 522 | yes |
| 5,0 | 146993,5 | 905 | 847 | yes |
| 6,3 | 80694,21 | 738 | 481 | yes |
| 7,0 | 153919,66 | 1206 | 1148 | yes |
| 8,0 | 123248,67 | 954 | 896 | yes |



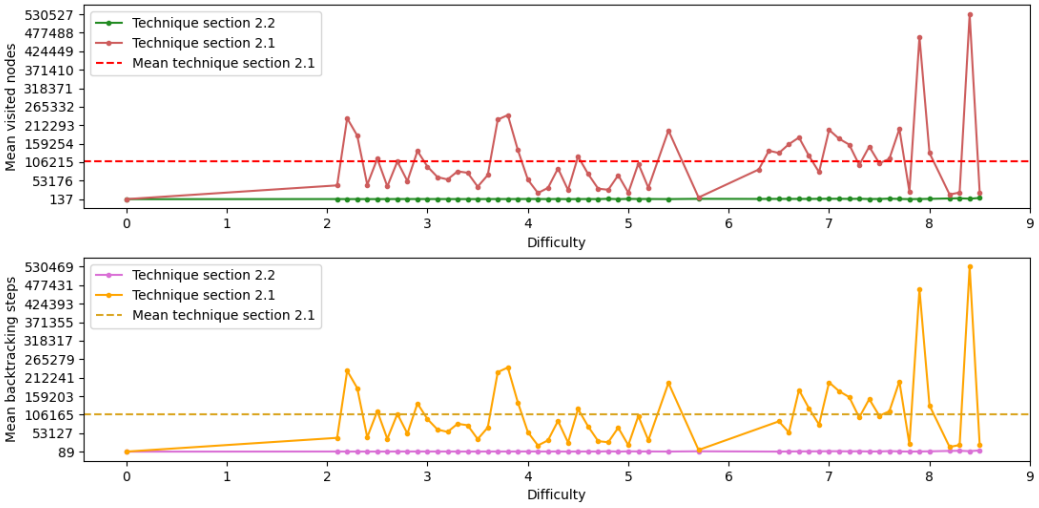
- Third and fourth approach addressed in the section 3.1 and 3.2 differ only in the execution time and therefore I preferred to use a unique table:

| Difficulty | Mean time ($\mu$s) technique 3.1 | Mean time ($\mu$s) technique 3.2 | Mean steps to converge | Solved |
|---|---|---|---|---|
| 0,0 | 69962868,93 | 34072743,41 | 225 | yes |
| 2,0 | 303592782,50 | 128692431,70 | 678 | no |



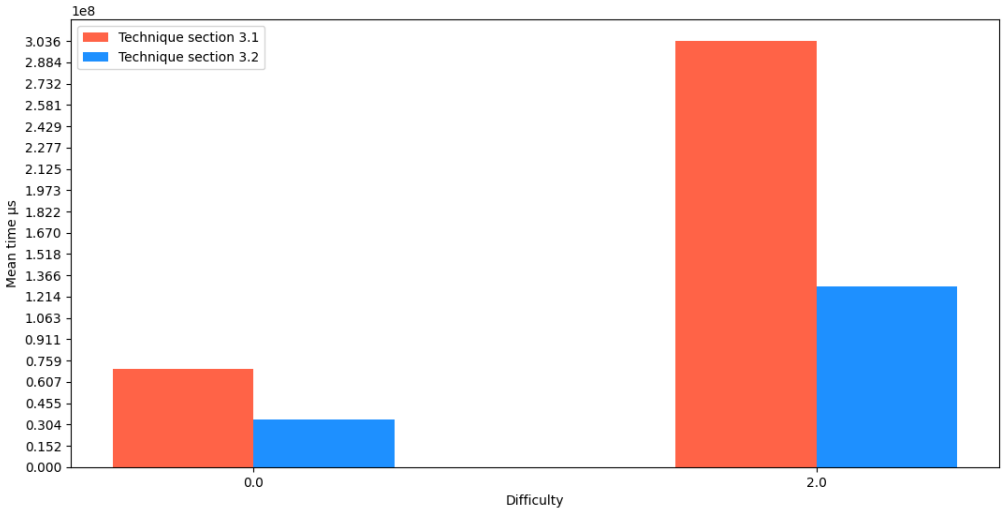
## 4.2 Comparison of all approaches

At the end of this report I want to make some reflections on the analyzed approaches in order to be able to deduce positive and negative aspects. As first comparison I want to analyze the techniques in sections 2.1 and 2.2, these two approaches both use Constraint Propagation and Backtracking and differ only in how the blank cell to fill is chosen. However, this has a significant impact on performance. In fact, as can be deduced from the graph below, both respond well for simple Sudoku, but increasing the difficulty you notice a sharp gap between the two approaches. In fact, the two techniques have two completely different scales.




Also in general known that the greater the complexity of the matrix the greater the number of tests that the algorithm must do, with the consequence of wasting time and resources with wrong choices. This effect is given by the backtracking strategy, which guarantees the resolution of Sudoku, but the disadvantage is that it wastes time in branches that are not useful. For this reason the addition of constraints improves the performance of the algorithm because the branches are deleted first.

As second comparison I want to discuss the techniques in sections 3.1 and 3.2, these two approaches both use Relaxation Labeling and differ only for type of implementation (sequential or parallel). In fact the second technique is more performant with regard to the aspect of the execution time, and this can be seen from the following graph. This approach is able to solve simple Sudoku, but for sudoku of greater difficulty spends much more time to give an answer, which will be wrong.




Now I want to discuss the difference between the two approaches, both try to solve the same problem, but the final conclusion is that the constraint propagation and Backtracking is able to find a solution regardless of the complexity of the matrix, Instead Relaxation Labeling solves only easy sudoku. In fact, the first approach ends when a solution is found, while the other approach uses a heuristic stop criterion associated with the convergence of the weighted label assignment and this does not always lead to a coherent solution. About the temporary complexity I can tell that Backtracking in the worst case explore all tree of possibilities $O(l^n)$, and for Relaxing Labels I can't specify any complexity since it continues to do a loop until happens a specific condition, but the complexity to make

a lap of loop is $\Theta(n^2 \times l^2)$. Regard spatial complexity Backtracking holds a matrix $n \times l$ so the complexity is $\Theta(n \times l)$, instead for Relaxing Labels it must hold the $P$ and $Q$ matrix, so the complexity is $\Theta(n^2 \times l^2)$ (in case that $Q$ isn't a matrix, and so using the function associated of it, the complexity decrease till $\Theta(n \times l)$).

$n$ represent the number of blank cells and $l$ represent the numbers that can be assigned.