Artificial Intelligence: Knowledge Representation and Planning

CM0472: Assignment 1 – Sudoku Solver

Team:

Ali Mohsen Alagrami - [**887727**]

Ivanoiu Alexandru Paul - [**870383**]

Index:

Assignment:

Write a solver for sudoku puzzles using a constraint satisfaction approach based on constraint propagation and backtracking, and one based on Relaxation Labeling.

Compare the approaches, their strengths, and weaknesses.

A sudoku puzzle is composed of a square 9x9 board divided into 3 rows and 3 columns of smaller 3x3 boxes. The goal is to fill the board with digits from 1 to 9 such that each number appears only once for each row column and 3x3 box; each row, column, and 3x3 box should contain all 9 digits.

The solver should take as input a matrix with where empty squares are represented by a standard symbol (e.g., ".", "_", or "0"), while known square should be represented by the corresponding digit (1, ... ,9).

For example: (the following example was given by our teacher on Moodle)

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
0	0	0	**1**	**5**	**4**	0	**7**	0
0	0	**3**	0	**7**	0	**6**	0	0
0	**5**	0	**6**	**3**	**8**	0	0	0
0	**6**	**4**	**9**	**8**	0	0	0	0
5	**9**	0	0	**2**	**6**	0	0	0
2	0	0	0	0	**5**	0	**6**	**4**

Hints for Constraint Propagation and Backtracking:

1. Each cell should be a variable that can take values in the domain (1, ..., 9).
2. The two types of constraints in the definition form as many types of constraints:

2.1. Direct constraints impose that no two equal digits appear for each row, column, and box.

2.2. Indirect constrains impose that each digit must appear in each row, column, and box.

3. You can think of other types of (pairwise) constraints to further improve the constraint propagation phase.
4. Note: most puzzles found in the magazines can be solved with only the constraint propagation step.

Hints for Relaxation Labeling:

1. Each cell should be an object, the values between 1 and 9 labels.
2. The compatibility $r_{ij}(\lambda, \mu)$ should be 1 if the assignments satisfy direct constraints, 0 otherwise.

Goal of this assignment:

The goal of this assignment is to insert any sudoku problem in input and to get the result in output using the 2 methods of solving mentioned above: first one being the **Constraint Propagation and Backtracking** and the second method being the **Relaxation Labeling.** After running them, we must decide which one is better and explain the ups and downs of each and one of them.

What is sudoku and how it works / how to play:

Sudoku is a board game first of all and it is played on a grid of 9 x 9 spaces. Inside the grid we have 9 smaller "squares" that are made of 3 x 3 spaces.

Each row, column and square have 9 spaces each and with that we mean that each column has 9 spaces, each row has 9 spaces, and each square has 9 spaces. Each space needs to be filled out with a number between 1 and 9 so each cell has a domain that goes from 1 to 9.

The golden rule/constraint is to not have two or more equal numbers on the same row / column and square, so each row has 9 spaces, and each space of that specific row has a different number from 1 to 9, same rule applies for the column and square.

The goal of this game is to assign to each space a number from 1 to 9 in such a way that each column has all the numbers from 1 to 9 without any repetition, each row has all the numbers from 1 to 9 without any repetition and the same goes for the square.

The grid that we take in input is filled with some numbers in some specific spaces.

0	0	0	2	6	0	7	0	1
6	8	0	0	7	0	0	9	0
1	9	0	0	0	4	5	0	0
8	2	0	1	0	0	0	4	0
0	0	4	6	0	2	9	0	0
0	5	0	0	0	3	0	2	8
0	0	9	3	0	0	0	7	4
0	4	0	0	5	0	0	3	6
7	0	3	0	1	8	0	0	0

Fig.1 Sudoku not solved

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

Fig.2 Sudoku solved

The more numbers we have when we start the easier it is and vice-versa.

The way you start solving the sudoku is by taking each individual cell and apply the row, column and block constraints which specify that you can put a number in a specific space just if that number is not found in the same row, column, or block. If the number, you want to insert has been used already you have to pick another one. At some point if you put a number in the wrong place you will be able to understand it and you must go back with the moves you made and change the numbers of the previous spaces you have filled.

Rules / Constraints:

We can say that sudoku is a problem that can be model using the **Constraint Satisfaction Problem** that we study during the course.

The **Constraint Satisfaction Problem** involves the following things:

1. A set of variables from 1 to n
2. The constraints that the variables must respect
3. For each variable we have a domain which contains the values that the variable can take

Once we mentioned the model and what the model involves, we should define more in details the constraints that we talked about above.

First constraint – Column constraint:

All the values on the same column must be distinct from each other and by that we mean that if we have the value "¥" [value between 1 and 9] inside a cell let's say "X_{ij}", at this point we have to say that for each other cell in that column the values of each cell must be different from "¥".

Second constraint – Row constraint:

All the values on the same row must be distinct from each other and by that we mean that if we have the value "¥" [value between 1 and 9] inside a cell let's say "X_{ij}", at this point we have to say that for each other cell in that row the values of each cell must be different from "¥".

Third constraint – Block/Square constraint:

In the entire grid we have 9 small squares or blocks that are formed of 3x3 spaces or cells. At this point let us say that we have a cell in one of the 9 blocks with the value "¥". Knowing this information, we are not allowed to insert or have other cells that contain that value so all cells in that square must have a distinct value.

Approach that we as a team followed for the first part of the assignment + code explained:

The first part is divided in two smaller parts:

1. Taking in input the grid and do the propagation
2. Once we have the domain of each cell "cleaned" we do the backtracking and propagating at each step

Code bellow:

```
def main():
    stringInputMatrix = "...26.7.1 68..7..9. 19...45.. 82.1...4. ..46.29..
.5...3.28 ..93...74 .4..5..36 7.3.18..."
    stringInputMatrix2 = "9.5...3.. ...1.3... ....257.4 .8..32.4. .54....8.
.1.58..6. ...4.1... .....3.59 8432....9"

    GRID = createMatrix(stringInputMatrix)

    # Now that we have the entire grip of numbers we need the domains of each
single cell
    Domains = createDomainMatrix()

    # Once we have the domain too we can print it
    print("####Domain before cleaning####")
    printDomain(Domains)

    # Once we have the domain we can do the first propagation to "clean" the
domain of each cell
    cleaningDomain(GRID, Domains)

    print("####Domain after cleaning####")
    printDomain(Domains)

    # Once I have the domain cleaned I can start working on that numbers to
change them and solve the sudoku
    backTracking(Domains, 0, 0)
```

Full explanation:

First thing we do on the first part is to take the string in input that contains the sudoku game.

Once we have the sudoku string we convert it into a matrix of integers.

Code:

```
# This function create the initial matrix starting from the input string
def createMatrix(stringInput):
    theGrid = [[-1] * 9 for i in range(0, 9)]
    stringInput = stringInput.replace(" ", "")
    for i in range(0, 9):
        for j in range(0, 9):
            if stringInput[i * 9 + j] != ".":
                theGrid[i][j] = int(stringInput[i * 9 + j])
            else:
                theGrid[i][j] = -1
    for arr in theGrid:
        print(arr)
    return theGrid
```

Since we talked about the domains until now, we create a separated matrix 9x9 and each element of the matrix has a domain that goes from {1 to 9} so we represent the domain of each cell from the initial matrix with another matrix what we will use later in the program.

Code:

```
def createDomainMatrix():
    matrixDomains = []
    for i in range(0, 9):
        matrixDomains.append([])
        for j in range(0, 9):
            matrixDomains[i].append([i for i in range(1, 10)])
    return matrixDomains
```

After the creation of the grid with the actual sudoku game and with the matrix containing the domain of each cell we call a function called **"cleaningDomain"** that does the propagation allowing us to clean the domains of the cells.

Code:

```
def cleaningDomain(matrix, domain):
    for i in range(9):
        for j in range(9):
            if matrix[i][j] != -1:
                domain[i][j] = [matrix[i][j]]
                domain = propagation(i, j, matrix[i][j], domain)
```

We do the propagation in the following way:

We start from cell (0,0) and we check if the cell contains a value, or it is empty which we encoded with -1.

If we see that we have a number that is different from -1 it means that the initial grid has a number inside that we cannot change, and the domain of that specific cell is exactly the value in the cell. Once we are sure about the that we propagate this value to the other cells from the same row, column, and block by deleting this value from their domains.

Code:

```
def propagation(specificRow, specificColumn, matrixValue, domain):
    # This is how I find the positions for each square
    startingPossitionRowBloc = specificRow - specificRow % 3
    startingPossitionColBloc = specificColumn - specificColumn % 3

    domain[specificRow][specificColumn] = [matrixValue]

    # Columns propagation
    for i in range(0, 9):
        if matrixValue in domain[specificRow][i]:
            if specificColumn != i:
                domain[specificRow][i].remove(matrixValue)

    # Row propagation
    for j in range(0, 9):
        if matrixValue in domain[j][specificColumn]:
            if specificRow != j:
                domain[j][specificColumn].remove(matrixValue)

    # Block propagation
    for o in range(startingPossitionRowBloc, startingPossitionRowBloc + 3):
        for p in range(startingPossitionColBloc, startingPossitionColBloc +
3):
            if matrixValue in domain[o][p]:
                if (o != specificRow) or (p != specificColumn):
                    domain[o][p].remove(matrixValue)
    return domain
```

After we do this operation for each cell, we know for sure we have a matrix domain that has just the right values for each cell and now we have to find which value from the domain of each cell is the perfect one for that specific position.

To do so we call the backtracking method and we developed it the following way:

If we are at the end of the matrix cell (8,8) we know for sure we finished and we return true.

If the domain of a cell is empty, we know that for sure we put a wrong number in a cell and we propagated the wrong numbers, so we return False to

stop the recursion and we go back to the previous domain we had before doing the last modifications. To go back to the previous domain we use a method of python called: ***copy.deepcopy(value)*** .

Let us say that a cell has a domain with more than one value at this point we take the first value of that domain and we say that the domain of that specific cell is the first value of it, and we delete the other values. Once we do that, we propagate this to the other cells from the same row, column and block and we go ahead like this until we get a false or a true.

Once we put all the value in the cell and we do not have domains that are empty we know we finished otherwise for sure some domain would be empty at some point.

Code:

```
def backTracking(domain, rows, cols):
    # I am at the end of the game
    if rows == 8 and cols == 8:
        print("Solution")
        for i in range(0, 9):
            for j in range(0, 9):
                print(domain[i][j], end=" ")
            print("\n")
        return True

    # I am at the end of a column and I have to go a next row
    elif cols == 9:
        backTracking(copy.deepcopy(domain), rows + 1, 0)

    # If I have an empty domain of a cell this means that at some point I put
a wrong number in a place so I have to
    # go back
    elif len(domain[rows][cols]) == 0:
        return False

    else:
        # If the domain has more than one value I do not know which is the
right one and I have to try them one by one
        # but I know that for that specific cell every value of the domain in
this moment is a valid value
        for h in domain[rows][cols]:
            backTracking(copy.deepcopy(propagation(rows, cols, h,
copy.deepcopy(domain))), rows, cols + 1)
```

1. Constraint propagation and backtracking – What it is and how it works:

To make the process easier and faster, we have to remove gradually the values that cannot be part of the final solution, this strategy is called **constraint propagation.**

In our specific case we eliminate the values from the domain of each cell in such a way that we always have inside the domain just the values that respect the constraints.

So, when a cell gets a specific value the algorithm has to propagate the constraints removing the value from the domain of each individual cell from the same row, column and block.

During the process we might insert a wrong number even if it respects the constraints and after a while, we understand that something is not write. At this point we have to go back to the previous steps and choose another number and go forward again. This is called **backtracking**.

Backtracking is an algorithm for solving problems recursively by trying to build a solution step by step, deleting the solutions that cannot satisfy the constraints of the problem.

In our specific case we are trying to fill the cells one by one, but once we find a number in a cell that does not lead us to a solution, we remove it by doing the backtracking and we try the next digit as we explained in our solution approach.

This approach is better than the guessing one because when it backtracks it drops a set of permutations that would not give the solution.

As we studied in class the best approach is to combine the constraint propagation and the backtracking in such a way that the propagation reduces the number of the states that can be explored deleting the branches that do not satisfy the constraints.

When the algorithm starts it would try to explore only the states that satisfy the constraints and once a branch does not lead to a solution it backtracks until it finds a consistent branch / state and tries another one.

References:

1. Slides provided in class
2. Google – Geekforgeeks website mostly
3. Book - (Prentice Hall Series in Artificial Intelligence) Stuart Russell, Peter Norvig - Artificial Intelligence_ A Modern Approach-Prentice Hall (2010)

2. Relaxation Labeling - What it is and how it works:

The first thing we should mention is that when you use **relaxation labeling** you will use the **labeling problem** and a good approach would be to use the **contextual information** since sometimes just the local information (previous part of the assignment used it) is **not** *enough*.

Since we mentioned a lot of terms, we should explain them.

Let us start with the **labeling problem**:

A labeling problem consists in the following:

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

The main goal is to assign a label of Λ to each object of B.

Contextual information is expressed in terms of a real-valued $n^2 \times m^2$ matrix of a compatibility coefficients:

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

The coefficient "$R = r_{ij}\ (\lambda\ ,\ \mu\ )$" measures the strength of compatibility between the two hypotheses "$\mathbf{b_i}$ is labeled $\boldsymbol{\lambda}$" and "$\mathbf{b_j}$ is labeled $\boldsymbol{\mu}$".

At the beginning of the game each object $b_i \in B$ is initialized with an m-dimensional (probability) vector:

$$\mathbf{p_i^{(o)}(\lambda) = (\ p_i^{(0)}(1), \ldots\ldots, p_i^{(0)}(m)\ )^T}$$

with $p_i^{(o)}(\lambda) >= 0$ and $\Sigma_\lambda\ p_i^{(o)}(\lambda) = 1$

The meaning behind the $p_i^{(o)}(\lambda)$ is: the probability of the object "i" that at time 0 is labeled as λ. Each and every single object has its own probability and own label.

By concatenating all this probabilities we obtain an initial weighted labeling assignment $p^{(0)} \in R^{nm}$.

The space of weighted labeling assignments is:

$$\mathrm{I\!K} = \Delta m = \Delta \times \ldots \times \Delta$$

where each Δ is the standard simplex of R^n.

The vertices of IK represent the unambiguous labeling assignments, but the main point of the relaxation labeling is to find a way to have no ambiguity.

The hole process starts with the initial labeling assignment and then using the relation techniques it starts reducing the ambiguity of the initial labels more and more until the probability is equal to 1 and the right label is positioned in the right cell.

There are some cases in which the relaxation labeling does not converge so as a result we **do not** have a final solution with no ambiguity of the labels for each object and that respect the constraints.

The next step it would be to update the probability of the vectors. To do so we use the formula that we saw in class introduced by Rosenfeld, Hummel, and Zucker in 1976:

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

Where

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

quantifies the support that context gives at time t to the hypothesis "b_i is labeled with label λ".

Another very important concept is the **Average Consistency**. The average local consistency of a labeling p ∈ IK is defined as follows:

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

At this point we have the following theorem:

If the compatibility matrix R is symmetric then any local maximizer p ∈ IK of A is consistent.

References:

1. Slides provided in class

Approach that we as a team followed for the second part of the assignment + code explained:

The main idea that we used is that at the beginning we have the labels (the labels go from 1 to 9) that we want to assign to each object (that is a cell) and we try to find the highest probability for each object so we know that if we have a bigger probability for the label "8" ad example for that cell we will assign it to that specific cell.

The problem that we encounter is that sometimes the relaxation labeling is not converging, like if we give in input some hard sudoku problems there is a high probability that it will not converge.

We proceed with the algorithm by updating the probabilities using the formulas that we discus in class (see above), and when we see that the change in the probability distribution that we have in a specific object does not change that much we assume that it converged(for this part we used the threshold as you will see below), and we assign the label with the highest probability for that object, but we are not guaranteed that is the right label 100%.

Full explanation:

1st function is the "**compatibility"** function that takes in input 2 positions and 2 values and it returns 1 if they are compatible and 0 if are not.

Code:

```
def compatibility(i, mi, j, lj):
    if mi == lj:
        if i % 9 == j % 9:
            return 0
        elif int(i / 9) == int(j / 9):
            return 0
        elif get_block(i) == get_block(j):
            return 0
        else:
            return 1
    else:
        return 1
```

2nd function is the "**create_compatibility**" function that creates a lookup matrix full of the compatibility of the cells. It is full of 0 and 1.

Code:

```
def create_compatibility():
    cm = np.zeros((81,9,81,9))
    for i in range(0,9*9):
        for mi in range(0,9):
            for j in range(0,9*9):
                for lj in range(0,9):
                    cm[i][mi][j][lj] = compatibility(i,mi,j,lj)
    return cm
```

3rd function is the "**get_block**" function that returns the block number so based on that I know where the row and cols start.

Code:

```
def get_block(i):
    block_n = 1
    if 0 <= int(i / 9) and 3 > int(i / 9):
        block_n *= 3 * 0
    elif 3 <= int(i / 9) and 6 > int(i / 9):
        block_n *= 3 * 1
    else:
        block_n *= 3 * 2

    if 0 <= i % 9 and 3 > i % 9:
        block_n += 0
    elif 3 <= i % 9 and 6 > i % 9:
        block_n += 1
    else:
        block_n += 2

    return block_n
```

4th function checks the consistency, is like checking how confident we are to assign that label to that specific object and at the same time we update the support.

Code:

```
def consistency_opt():
    update_support_opt()
    tot_cons = (support * p).sum()
    return tot_cons
```

5th function is the "**init**" function that creates the compatibility matrix and creates the initial probabilities, as we can see we have **"for i in range(0,9*9)** " so we loop over all the cells and for each cell we create a probability.

Code:

```
def init(sk):
    global p,support,cm
    cm = []
    p = np.empty((0,9))
    support = []

    cm = create_compatibility()

    from scipy.stats import norm
    for i in range(0,9*9):
        row = np.linspace(norm.ppf(1/10), norm.ppf(1/9), 9)
        p = np.append(p, np.array([row/row.sum()]),axis=0)

    support_normal = [[0]*9 for i in range(9*9)]
    support = np.asarray(support_normal)

    for i in range(0,len(sk)):
        for j in range(0,len(sk)):
            if sk[i][j]!=-1:
                p[i*9+j][sk[i][j]]=1
                for v in range(0,9):
                    if v != sk[i][j]:
                        p[i*9+j][v]=0
```

6th function is the "**check**" function that gives me the number of errors that we have inside the matrix. So if we have 2 cells inside the same column both with the same number we have 2 errors since there are two cells that have the same number in the same column.

Code below:

```
def check(sk):
    errors = 0
    for i in range(0, 9):
        errors += sk[i][:].size - np.unique(sk[i][:]).size
        errors += sk[:][i].size - np.unique(sk[:][i]).size
    for i in range(0, 9):
        for j in range(0, 9):
            for ki in range(0, 9):
                for kj in range(0, 9):
                    if ki != i and kj != j:
                        if sk[i][j] == sk[ki][kj] and get_block(i * 9 + j) ==
get_block(ki * 9 + kj):
                            errors += 1

    return errors
```

The last function is the "**solve**" that does the following:

At the beginning it creates the matrix using the "init_matrix" function that takes in input the string of the sudoku game.

Once we have created the sudoku matrix we create the "p_count" variable so we can print each 100 iterations the consistency or the probability or whatever we want to print.

Then we have the "thre" variable that stands for the threshold and is calculated between the change in the probability distribution in the current iteration and on the previous one.

The while function allows us to do check if the actual threshold is bigger than my threshold and if it smaller it gets out of the cycle otherwise it enters and does the iterations.

Inside the while we update the consistency, we update the probability and we save the old probability and then we use it to calculate the new threshold using the method "linalg.norm" taking as parameters the actual probability of this iteration and the old one.

Code:

```
def solve(arr,my_thre,count):
    current = 0
    peak = 0
    p_count = 100
    sk = init_matrix(arr)
    init(sk)
    thre = 1
    while thre>my_thre and count>0:
        if p_count == 0:
            print(thre)
            p_count=100
        else:
            p_count-=1
        count-=1
        old_p = p
        consistency_opt()
        update_prop_opt()
        thre = np.linalg.norm(p-old_p)
    return return_sol(sk)
```

Comparison between "Constraint propagation and backtracking" and "Relaxation Labeling" :

Above we saw two ways to solve a sudoku, the first way was the **Constraint propagation and backtracking (CPB)** and the second one was the **Relaxation Labeling (RL)**. The most important difference that we observed during development is that the **CPB** finishes every time independently of the difficulty of the sudoku, so it stops when it finds a solution meanwhile the **RL** manages to solve just the easy once so not always finishes.

Below we can the time comparison between the two:

1. CPB execution time: **0m3.372s**
2. RL execution time: **1m2s**

To have a fair comparison between the two we use the 4 main criteria that we studied: Completeness – Optimality – Time complexity – Space complexity.

Completeness:

1. CPB: Yes, the CPB is complete since it guarantees to find a solution.
2. RL: No, the RL it not complete since you do not have a guarantee that it will finish and find the right solution since it can solve only the easy once.

Optimality:

1. CPB: We cannot say that is optimal but is the best one for the minimum number of steps
2. RL: It is not optimal because as we already said it can solve just the easy once

Time complexity:

1.CPB: O (m^n)

2.RL: O (n^2 x m^2)

Space complexity:

1. CPB: O (n)
2. RL: O (n^2 x m^2)