Università
Ca'Foscari
Venezia

[CM0472-1] Artificial intelligence: knowledge representation and planning

First assignment

Sudoku solver

Author:
Santello Veronica, 870320
Rossato Davide (collaboration only on code)

Academic Year:
2020/2021

Due date:
22/3/2021

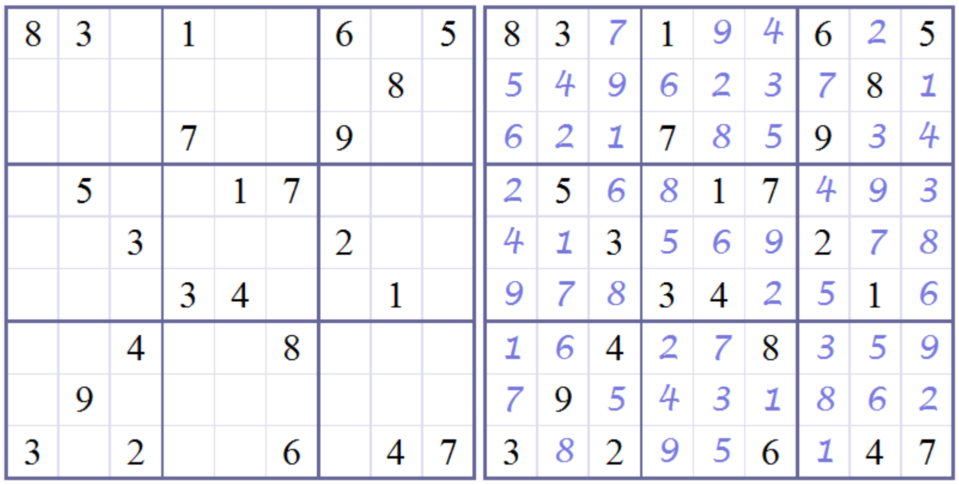
Contents

1 Introduction

A Sudoku puzzle is composed of a square 9x9 board divided into 3 rows and 3 columns of smaller 3x3 boxes. The **goal** of this game 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 contains all digit between 1 and 9

At the begging, the puzzle is given to the player partially completed (as we can see on the left box in fig:1) and thus the goal for him is to complete the rest of cells in order to satisfy all the rules. An example of Sudoku resolution is given in the second box of the fig:1.



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

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

Figure 1: Example of Sudoku resolution [1]

To solve a Sudoku, there are different techniques depending on the difficulty of the Sudoku itself. Normally, the Sudoku must have only one solution; to be sure of this, the puzzles are presented with a number of digits already present in the initial grid, leaving the player to deduct the remaining digits to be entered in the free cells. *"If a Sudoku has only one solution, it has been proved that there must be at least 8 of the 9 digits present in the initial grid as entries. If only 7 numbers are given, then the puzzle has at least two solutions."*[2]

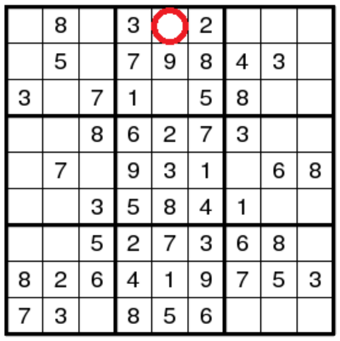
To find the solution, a brute force approach is not an efficient idea in terms of time and space, since the total number of states produced by a Sudoku is very high. The main idea is search only for states that can reach the solution, and eliminate the others.

2 Constraint satisfaction problems

Constraint satisfaction problems(CSPs) are mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. [4]

The Sudoku problem can be formally defined as a CSP as:

- A set of variables:
$$X = x_1, ..., x_{81}$$
In our case a variable is a single cell of the game board.
- $\forall x_i$ is associated a domain set D_i which contains the possible values that x_i can assume. In our case each domain contains all values in the range: $[1,2,3,4,5,6,7,8,9]$.
- A set of constraints:
  - **Row constraint**: all values in the row i must be different, $\forall i \in 1, ..., 9$.
  - **Column constraint**: all values in the column j must be different, $\forall j \in 1, ..., 9$.
  - **Box constraint**: all values in the box k must be different, $\forall k \in 1, ..., 9$.
  - **Indirect constraints**: impose that each digit must appear in each row, column, and box. This particular constraint consists to take advantage of filled cells to restrict if it is possible the domain set of empty cells.
    For example:the cell $x_{1,5}$ marked with the red circle in the figure 2 has an initial do-



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

Figure 2: Example indirect constraint [5]

main $D_{1,5} = 1,2,3,4,5,6,7,8,9$, but if we take into account the previous constraints we are able to define a more accurate domain, that is $D_{1,5} = 4,6$.

This type of constraint improves the performance of the algorithm to find the solution and reduce the number of guess.

3 Constraint propagation and backtracking

The first point of the assignment consists of write an algorithm that uses both the constraint propagation and the backtracking method to compute a solution for an input Sudoku matrix 9x9.
Before starting to explain my solution, I want to give you a brief explanation of what backtracking and the constraint propagation are.

The **constraint propagation strategy** is based on the idea that for each domain, the algorithm removes the values that will not satisfy the constraints properties for the game seen in the previous section. In practice, when a cell is filled with a value i, the algorithm remove the value i from all interested domains (on the same row, column and box).

The **problem** arises when entering a value into a cell $x_{i,j}$ is not a secure step, since other values can be possible solution for $x_{i,j}$.The choice may be fine for some iterations but at certain point, it is possible that you have to go back and change the value, since there is no solution with this configuration of entered values.

The **backtracking** strategy takes over at this point. *"It is a type of brute force search"*[6] used for problems that generate a large state space tree, in which each node represents a possible state of the problem. The backtracking algorithm visit this tree in a *dept-first-search* order to find the solution of the problem. If in a certain moment the path taken not reflect the right way to find the solution, it goes back to the previous step and chooses a different path. In the figure 3 there is an example of tree states expansion.

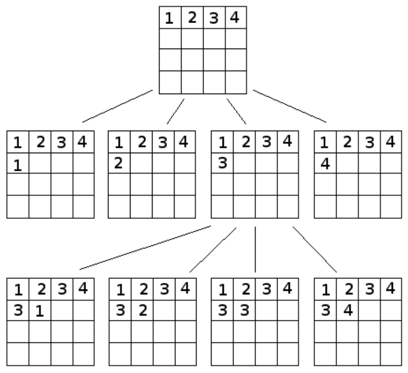


Figure 3: Example of backtracking tree [3]

Since using only the constraint propagation not guaranteed to find a solution and the backtracking algorithm has a large time complexity, since basically explore all possible state, **the better choice is combining these two strategies**. The constraint propagation reduces the number of explored states with the backtracking strategy, eliminating the useless breaches of the tree state.

3.1 Implementation

First the algorithm start his computation, in the file `main.py`, the player choose the matrix board 9x9 (or write it) in matrix form and call the function `functions.define_domains(matrix)` that returns a set of domains, one for each cell of the board. At this point, the player can calls the `solve` function. Obviously the initial domains of the already filled cells are initialised only with their value already entered.

```python
def solve(matrix, dom):
    find = find_empty(matrix)
    if not find:
        return True
    else:
        row, col = find
    if [] in dom:
        return False

    for i in dom[row*9 + col]:
        matrix[row][col] = i
        if solve(matrix, modify_domains(matrix, copy.deepcopy(dom), i, (row, col))):
            return True

        matrix[row][col] = 0

    return False
```

This is the heart function of my program that works like this: every recursion environment has as parameters the matrix updated and mostly the set of domains updated. After checking the **base cases**, that is:

- If there isn't an empty cell in the matrix, the solution is found and the algorithm return true.

- An another important case is when at least a domain in the set is empty. It means that in a previous recursion, some values filled have caused changes to some domains that cannot lead to a valid solution, since for at least one cell there are no possible value to enter. In this case the algorithm return false and doesn't continue the recursion in that branch.

Instead if there are at least an empty cell and the set of domains is valid, the algorithm can continue its computation in that branch, trying to insert in the empty cell found one of its possible values in the domain, and then call the recursion.

The parameters of the recursion call are the matrix, modified with the filled value just inserted, and the list of domains, modified with the function `modify_domains`.

```
def modify_domains(matrix, dom, num, pos):
```

This function applies the indirect constraint with the constraint propagation strategy, eliminating the value just inserted `num` in position `pos` from the list of interested domains `dom`.

3.2 Performance

Some medium measurement after some tests:

Difficulty Sudoku	N° Recursions	Time in seconds	Solved
Easy	55	0.062	Yes
Hard	2600	2.56	Yes

4 Relaxation Labeling Algorithm

The second point of the assignments consists of not use only local information but also contextual information in the so called relaxation labeling algorithm.

"*Relaxation labeling processes are a popular class of parallel, distributed computational models aimed at solving (continuous) constraint satisfaction problems, instances of which arise in a wide variety of computer vision and pattern recognition tasks*".[10]

Our labeling problem is defined by:

- A set of n =81 objects $B = b_1, ..., b_{81}$ that represented the 81 cells of the Sudoku board.
- A set of m =9 labels $\Lambda = 1, .., 9$ that represents all the possible digits to enter in each cell.

To accomplish this, two sources of information are exploited. The first one relies on local measurements which capture the salient features of each object viewed in isolation; classical pattern recognition techniques can be practically employed to carry out this task. The second source of information, instead, accounts for possible interactions among nearby labels and, in fact, incorporates all the contextual knowledge about the problem at hand.[10]

The goal is assign a label of Λ to each object in B. Contextual information are represented as compatibility coefficients

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

which measures the strength of compatibility between the two hypothesis "b_i is labeled with λ" and "b_j is labeled with μ".

The compatibility function in the file `functions1.py`

```
def compatibility(i, j, lb, mu):
    #if the objects are equal return zero
    if i == j:
        return 0
    if lb != mu:
        return 1
    if valid(i, j) == 0:
        return 0
    return 1
```

is implemented in order to return a 1 if the assignment is valid, 0 otherwise.

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

$$p_i^{(0)}(\lambda) = (p_i^{(0)}(1), ..., p_i^{(0)}(9))^T$$

with $p_i^{(0)}(\lambda) \geq 0$ and $\sum_\lambda p_i^{(0)}(\lambda) = 1$ where $p_i^{(0)}(\lambda)$ is the probability distribution at time 0, that the object i is labeled with the label λ.For simplicity we have assumed that the initial probability is computed with the use of the initial list of domains explained in the previous section [3] .

For example: if the first cell of the matrix has initial domain equal to [1,2,3,4], then

$p_1^{(0)}(\lambda) = (0.25, 0.25, 0.25, 0.25, 0, 0, 0, 0, 0)$. Each object is associated with to one probability distribution, and the union of these defines a weighted labeling assignment $p^{(0)} \in \mathbb{R}^{nm}$.

All the possible weighted labeling assignment belong into a space $\mathbb{K}$ so defined:

$$\mathbb{K} = \Delta^{\mathrm{m}} = \Delta \times \ldots \times \Delta$$

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

Each vertex of $\mathbb{K}$ represents an unambiguous **labeling assignment**, that is one which assigns exactly one label to each object. *"The task of relaxation labeling is to iteratively eliminate extra labels at ambiguous locations, so that at the end of processing only one label remains"*.[7]

This ambiguity is reduced considering the compatibility r_{ij} previous defined between objects and labels. The idea is that at the end of the iterative procedure all the probability distributions have an high probability, ideally equal to 1 but sometimes a little bit less, to the right label that should be positioned in that cell.

Relaxation strategies do not necessarily guarantee convergence, and thus, the procedure may not arrive at a final labeling solution with a unique and unambiguous assignment of the labels for each object and that satisfy game constraints. In fact, at the end of the iterative procedure it's possible that some vectors provide ambiguity or inconsistent assignments.

Let's see some examples with the object b_1 at time t:

- if $p_1^{(t)}(\lambda) = (0,0,1,0,0,0,0,0,0)$ there isn't no ambiguity and it will labeled with 3.
- if $p_1^{(t)}(\lambda) = (0,0,0,0.9,0,0,0,0.1,0)$ there isn't no ambiguity and it will labeled with 4.
- if $p_1^{(t)}(\lambda) = (0,0,0.5,0.5,0,0,0,0,0)$ there is ambiguity since we don't known if the right label is 3 or 4.

Each iteration step updates the vectors probability using the following heuristic formula, provided by Rosenfeld, Hummel and Zucker in 1976:

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

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

[8]

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

The classical stopping criteria is the **Average Local Consistency**, but we won't use it.

It is not recommended because, since our aim is to maximise the objective function, it would measure the increment of the function too fast since it works with the outputs only and does not take into account the input.For this reason, we will instead use the **Euclidean distance** applied to the probability vector and therefore to the convergence results of our algorithm. The formal formula of the Euclidean distance is:

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

where $P = (p_1, ..., p_n)$ and $Q = (q_1, ..., q_n)$ [9] represent the current probability vector and the probability vector of the previous step.

At each step the euclidean distance is computed and the algorithm stops when it reaches a specific constant value called **threshold** which is 0.001. When the threshold is reached also the convergence is reached.

4.1 Implementation

The main function is in the file `functions1.py` and is called `solve`. It takes in input the 9x9 matrix, and the list of probability distributions p.

```python
def solve(matrix, p):
    start = time.time()
    diff = 1
    iterations = 0
    old_p = copy.deepcopy(p)
    while diff > 0.001:
        compute_all_q(p)
        compute_all_p(p)
        diff = compute_distance_euclidean(p, old_p)
        old_p = copy.deepcopy(p)
        iterations += 1
        print("Actual difference : ", diff)

    print("---------FINAL BOARD----------")
    functions.print_board(choose_values(matrix, p))
    print('\n')
    print('\n')
    print("---------REPORT----------")
    print("TIME IN SECONDS %s :" % (time.time() - start),
    "NUMBER OF ITERATIONS :", iterations)
```

During the while cycle the distance euclidean is computed between the input probability vector and the output probability vector. At the end the function `print_board` has a special input that is the result of the function `choose_values` that labeled every cell of the matrix with the corresponding label having higher probability in it's distribution.

4.2 Performance

Some medium measurement after some tests:

Difficulty Sudoku	**N° Recursions**	**Solved**
Easy	160	Yes
Hard	452	No

The time in this solution is not optimised but falls within the average time discovered by some scholars, which is about **55 seconds**.[11]

5 Comparison

	Backtracking	**Relational labeling**
Completeness	Yes, since it guarantees to find a solution if exists	No, since it solves only easy Sudoku
Optimality	Yes, since if we assume that a solution exists, than it will always provides it with the minimum number of steps	No, since it solves only easy Sudoku
Time complexity	$O(m^n)$	$O(t \times m^5)$
Space complexity	$O(n)$ for the matrix , $O(n \times m)$ for the domains	$O(m \times n)$ for p and q
Stopping criteria	Stop when find a solution, or when the state space tree is fully visited.	Using an heuristic stopping criteria with the use of Euclidean distance.

Where n = 81 is the number of objects and m=9 is the number of labels.

I want specify that, in the time complexity of relaxation labelling, the heaviest function is `compute_all_q()` and so the total complexity assumes its complexity that is $O(t \times m^5)$where t is the unknown number of `while` cycles . A possible improvements, in terms of calculation efficiency in Python for the relaxation labeling, can be the computation of all compatibility at once and not every time it is asked.

6 Conclusions

In this assignment are considered two strategies to solve the Sudoku game. The first one defines an algorithm that solves it using constraint propagation and backtracking technique in order to provide completeness property. The second one considers Sudoku as a labeling problem, and tries to take advantage on the contextual information, in addition to the local ones, and applies iteratively an heuristic formula to update the current state until a stopping criteria is reached.

As we have seen the first one will always find a solution, if it exists, but time complexity increases with the complexity of the puzzle.

The relaxation labeling algorithm instead works well if the puzzle is not so difficult, since it is able to solve only easy configurations. If the rating increases the algorithm is not always able to take the right decision and updates on the right way the states. This approach seems to be not so efficient for this kind of problems.

We can conclude that the backtracking algorithm is more accurate and complete than the relaxation labeling for this problem, the only weakness is the time complexity that increase with the difficulty of the input matrix.

References

[1] *Source image:* https://codinghelmet.com/exercises/sudoku-solver

[2] *Source:*http://quantidiscienza.blogspot.com/2016/12/la-matematica-del-sudoku.html

[3] *Source image:*https://bitcook.de/backtracking-method/

[4] *Source:* https://en.wikipedia.org/wiki/Constraint_satisfaction_problem

[5] *Source image:* https://homepages.cwi.nl/~aeb/games/sudoku/solving28.html

[6] *Source:* https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

[7] *Source:* http://www.bcp.psych.ualberta.ca/~mike/Pearl_Street/Dictionary/contents/R/relaxationlab.html

[8] *Source:*https://www.dsi.unive.it/~pelillo/Didattica/Artificial%20Intelligence/Old%20Stuff/relaxation%20labeling.pdf

[9] *Source:* https://it.wikipedia.org/wiki/Distanza_euclidea

[10] *Source:*https://www.dsi.unive.it/~pelillo/papers/PatternRecognition%202000.pdf

[11] *Source:*https://www.slideserve.com/delano/sudoku-via-relaxation-labeling-powerpoint-