Assignment 1: Sudoku Solver

Francesco Maria Trasforini, student ID: 887156

March 21, 2021

Contents

1 Introducing the game of Sudoku **1**

2 Constraint Propagation and Backtracking **2**

3 Constraint Propagation and Backtracking - Implementation **3**

4 Relaxation Labeling **4**

5 Relaxation Labeling - Implementation **6**

6 Time Performance Comparison **8**

7 Conclusions **10**

Chapter 1

Introducing the game of Sudoku

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

1. each number appears only once for each row, column and 3x3 box;
2. each row, column, and 3x3 box should containg all 9 digits.

Sudoku puzzles come with some filled cells. Those cells cannot change values and act as base constraints of the puzzle.

Chapter 2

Constraint Propagation and Backtracking

The rules listed above suggest that Sudoku is a game well suited to be modelled as a **Constrain Satisfaction Problem**:

- any of the empty cells is a variable. The set of variables is $X = \{x_1, x_2, ..., x_n\}$, where n is the number of empty cells
- Given $C_{row} = \{constraints_{row}\}, C_{col} = \{constraints_{column}\}, C_{box} = \{constraints_{box}\}, \forall x_i \in X, \exists D_i \subseteq D = \{1,2,3,4,5,6,7,8,9\} : D_i = D \setminus C, C = C_{row} \cup C_{col} \cup C_{box}$.
  In other words, any of the n cell has its own **domain**, which is easily computed by subtracting all the active constraints in the given position from the naive domain set D

Constraint propagation takes an approach not dissimilar to brute force in principle. The reason why constraint propagation works much better is simple: by exploiting the constraints available, it reduces significantly the search space.

The proposed implementation combines backtracking and constraint propagation. The table is scanned by row starting from the top left corner. Given the constraints, if there are possible assignments for the current cell, one of them is selected (in my code, I choose to select the smallest available value). When a cell is filled with a new value, the constraints for the current row, column and box are updated, ensuring that the insertion of future values will not break the game rules. If there are no possible assignments for the current cell, the algorithm backtracks to the previous step and tries a different path.

Chapter 3

Constraint Propagation and Backtracking - Implementation

```python
def solve_cpb(board):

    empty_position = find_empty(board)
    if not empty_position:
        return True
    else:
        row, col = empty_position

    for k in range(1,10):
        if (check_constraints(board,k,row,col)):
            board[row][col] = k
            if(solve_cpb(board)):
                return True
            board[row][col] = 0

    return False
```

Figure 3.1: Constraint Propagation and Backtracking solver function

The solver starts by looking for an empty position on the board. As soon as it finds one, that position becomes the active one. If there are valid assignable values for the current position, the solver tries to assign to it the smallest of those values and calls itself recursively. If there are no possible valid assignments for a cell, the algorithm backtracks to the previous state and tries a different assignment. The execution stops when the search for empty values fails: this event signifies that the board has been completely filled and the solution has been reached.

Chapter 4

Relaxation Labeling

Relaxation Labeling takes a different approach from Constraint Propagation and Backtracking.
In general, a labeling problem is defined by:

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

In the game of Sudoku:

- $\Lambda = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- B = {empty cells on the board}

The goal is to assign a label to each object.
Relaxation labeling uses 2 sources of information:

1. local measurements to capture salient features of each object viewed in isolation
2. contextual information expressed in terms of a real valued n2 x m2 matrix of compatibility coefficients $R = \{r_{ij}(\lambda, \mu)\}$.
   r_ij is a coefficient measuring the strenght of compatibility between the two hypothesis:
   - b_i is labeled λ
   - b_j is labeled μ

The compatibility coefficient r_{ij} is equal to 1 if the assignment is valid, it is equal to 0 if it is not.

At startup, $\forall b_i \in B$ an m-dimensional probability vector of initial local measurements

$$p_i^{(0)}(\lambda) = (p_i^{(0)}(1), ..., p_i^{(0)}(m))^T, p_i^{(0)}(\lambda) \geq 0, \sum_{\lambda} p_i^{(0)}(\lambda) = 1 \tag{4.1}$$

is provided.
The vectors $p_i^{(0)}(\lambda)$ are representations of the initial noncontextual degree of confidence in the hypothesis b_i is labeled λ.
To obtain the initial weighted labeling assignment $p^{(0)} \in \mathbb{R}^{nm}$ it is sufficient to concatenate the vectors $p_1^{(0)}, ..., p_n^{(0)}$.
The space of weighted labeling assignments is
$IK = \Delta^m = \underbrace{\Delta * ... * \Delta}_{\text{m}}$, where each Δ is the standard simplex of $\mathbb{R}^n$.

$$IK = \{\bar{p} \in \mathbb{R}^{nm} : \sum_{\lambda=1}^{m} p_i(\lambda) = 1, i = 1, ..., n, p_i(\lambda) \geq 0, i = 1, ..., n, \lambda \in \Lambda\} \tag{4.2}$$

Each vertex of IK gives an unambiguos labeling assignment. In other words, it assigns one label to each object. A relaxation labeling algorithm takes as input $p^{(0)}$ and it iteratively updates it in accordance with the compatibility matrix R. Relaxation labeling does not guarantee convergence. At the end of the iterative process, it may happen that some of the assignments do not act accordingly to the game rules. For a relaxation labeling algorithm, the ideal would be to have probability distributions equal to 1 $\forall$ right couple (object, label) at the end of the process.

At any step, the vector probability is updated following the (heuristic) rule introduced by Rosenfeld, Hummel, Zucker in 1976:

$$p_i^{t+1}(\lambda) = \frac{p_i^t(\lambda) q_i^t(\lambda)}{\sum_{\mu} p_i^t(\mu) q_i^t(\mu)} \tag{4.3}$$

$q_i^t(\lambda)$ is a measure of the support given to the hypothesis "b_i is labeled λ by the current context.
The exact expression of $q_i^t(\lambda)$ is:

$$q_i^t(\lambda) = \sum_j \sum_{\mu} r_{ij}(\lambda, \mu) p_j^t(\mu) \tag{4.4}$$

The local average consistency of a label $p \in IK$ is defined as:

$$A(p) = \sum_i \sum_{\lambda} p_i(\lambda) q_i(\lambda) \tag{4.5}$$

The main weaknesses of relaxation labeling are:

1. inability to guarantee convergence
2. stopping criteria not formally defined

Chapter 5

Relaxation Labeling - Implementation

I implemented the algorithm with matrix notation.

```
def relaxationLabeling():
    global rij, p
    diff = 1
    avg_b = 0
    t = 0
    while diff > 0.001:
        q = np.dot(rij, p)
        num = p * q
        row_sums = num.reshape(ncells*ncells,ncells).sum(axis=1)
        p = (num.reshape(ncells*ncells,ncells)/row_sums[:, np.newaxis]).reshape(729,1)
        avg = averageConsistency(q)
        diff = avg - avg_b
        avg_b = avg
        t += 1
    p = p.reshape(totcells, ncells)
```

Figure 5.1: Relaxation Labeling function

I choose to stop the iterations when the difference between the average consistency at step t (the current step) and the average consistency at step t-1 (the previous step) becomes smaller than 0,001.

$$A_t(p) - A_{t-1}(p) \geq 1 \tag{5.1}$$

To this end, it is important to note that it is guaranteed that following the Rosenfeld, Hummel, Zucker update rule, the average consistency strictly grows.

```
def solve_relaxationLabeling(board, create = True):
    global ncells,totcells, p, rij
    if (create):
        initializeRij() # initialize matrix Rij at startup
        create = False
        p = np.ones((totcells*ncells, 1))/ncells

    ncells = len(board_c)

    for row in range(ncells):
        for col in range(ncells):
            domain = np.array([1,2,3,4,5,6,7,8,9])
            row_container = np.array([],int)
            col_container = np.array([],int)
            for k in range(9):
                row_container = np.append(row_container,board_c[row][k])
                col_container = np.append(col_container,board_c[k][col])
            #eliminate 0s from row_container and col_container
            row_container = row_container[row_container != 0]
            col_container = col_container[col_container != 0]
            #I find the box of the current position
            box = box_placement(row,col)
            #box_container is a vector where I store the values currently inserted in the given box
            box_container = box_content(box,board_c)
            #eliminate 0s from box_container
            box_container = box_container[box_container != 0]
            constraints = np.concatenate((row_container, col_container, box_container))
            constraints = np.unique(constraints)
            valid = np.setdiff1d(domain,constraints)
            n = len(valid)
            prob = np.zeros((1, ncells))[0]
```

```
            if (not board_c[row][col] == 0):
                # value just assigned
                val = int(board_c[row][col])
                prob[val-1] = 1
            else:
                for k in valid:
                    prob[int(k) - 1] = 1/n + randint(0,20)/100.0
            prob = prob/np.sum(prob)
            p.reshape(ncells,ncells,ncells)[row][col] = prob


    rij = np.loadtxt("rij.csv", delimiter=",")
    relaxationLabeling()
    for i in range(totcells):
        pos = np.argmax(p[i])
        if i % ncells == 0:
            print("")
        print(pos+1, end=" ")
```

Figure 5.2: Relaxation Labeling solver function

Chapter 6

Time Performance Comparison

To test the 2 algorithms I used a dataset of easy Sudoku found on Kaggle (https://www.kaggle.com/bryanpark/sudoku). I run my test over 100 puzzles.

```
import time as t
exec_time = np.zeros((r,2))
for i in range (0,r):
    b = np.zeros(81,dtype=np.int32)
    b = prep_data(i)
    new_arr = b.reshape(9,9)
    board = new_arr
    board_c = np.copy(board)
    start = t.time()
    solve_cpb(board)
    end = t.time()
    exec_time[i][0] = end - start
    start = t.time()
    solve_relaxationLabeling(board_c)
    end = t.time()
    exec_time[i][1] = end - start
```

Figure 6.1: Run the algorithms and store the running time of each iteration

The matrix exec_time is a matrix of size (r,2), where r is the size of the test set (100 in the test shown). In the first column I stored the execution times of the Constraint Propagation and Backtracking algorithm, in the second one the execution time of the Relaxation Labeling algorithm. So, in position (0,0) there is the running time for the execution of Constraint Propagation and Backtracking over the first puzzle and in position (0,1) there is the running time for the execution of Relaxation Labeling over the same puzzle.
This simple structure makes it easy to compute the mean running time for each algorithm.

```
print("Average execution time\nConstraint Propagation and Backtracking - Relaxation Labeling: ",np.mean(exec_time, axis=0))
```

```
Average execution time
Constraint Propagation and Backtracking - Relaxation Labeling:  [0.54153098 3.07682598]
```

The running time standard deviation is

```
print("Execution time standard deviation\nConstraint Propagation and Backtracking - Relaxation Labeling: ",np.std(exec_time, axis
```

```
Execution time standard deviation
Constraint Propagation and Backtracking - Relaxation Labeling:  [0.44643146 1.25915054]
```

The figure below shows the ratio between the mean running time of the 2 algorithms:

```
np.mean(exec_time[:,1],axis=0)/np.mean(exec_time[:,0],axis=0)
```

```
5.681717360523512
```

Chapter 7

Conclusions

There are no doubts that the best Sudoku solver among the 2 proposed is the Constraint Propagation and Backtracking one. The first thing to remark is that Constraint Propagation and Backtracking is guaranteed to find a solution, while Relaxation Labeling is not. In other words, Constraint Propagation and Backtracking is a Complete algorithm, Relaxation Labeling is not.

In general, the temporal complexity is:

1. $O(m^n)$ for Constraint Propagation and Backtracking. Pruning the invalid branches (the ones breaking the constraints) is crucial to bring down the execution time.

2. $(n^2 \text{ x } m^2)$ per step for Relaxation Labeling. The overall time complexity is not well defined because it depends on the stopping criteria.

It is worth noticing that even in situations in which Relaxation Labeling is able to reach a solution, it is much slower than Constraint Propagation and Backtracking, as shown in the Time Performance Comparison chapter. This means that Sudoku is a problem in which local information has a dominant importance over contextual information. Relaxation Labeling performs best when it is applied to solve problems in which contextual information is relevant. Examples of such problems can be found in the fields of Computer Vision and Pattern Recognition.