Ca' Foscari University of Venice

ARTIFICIAL INTELLIGENCE: KNOWLEDGE REPRESENTATION AND PLANNING

[CM0472-1]

Assignment 1: Sudoku Solver

Report by:
Zonelli Mattia, 870038

Code by:
Zonelli Mattia, 870038
Garbin Eleonora, 869831

Contents

1 Introduction

A Sudoku is a puzzle game based on a grid of 9x9 cells that can be split up in 9 "boxes" of 3x3 cells. The goal is to assign to each cell a value between 1 and 9 such that in each row, column or box each digit appears exactly once. The grid is usually pre-filled with some numbers and the player has to fill all the others cells such that the rules are satisfied. At each starting configuration correspond a different and sometimes unique solution, the less the number of pre-completed cells the more complex the Sudoku will be. A Sudoku has a very large of possible states, and it can be seen as search problem. To solve a Sudoku is necessary to reduce the possible values of each cells down to one. Usually the player has to take advantages of the constraints, rules of the game to reduce the search space.

In our implementation of the Sudoku, empty cells are represented by cells with value "0", while cells with assigned value is represented by corresponding digit.

```
[7, 0, 0,   0, 0, 9,   0, 0, 6],
[0, 0, 0,   0, 0, 0,   0, 0, 0],
[5, 0, 8,   4, 1, 0,   0, 2, 0],

[0, 0, 1,   0, 6, 0,   0, 0, 0],
[0, 5, 2,   0, 0, 0,   7, 8, 0],
[0, 0, 0,   0, 5, 0,   3, 0, 0],

[0, 9, 0,   0, 7, 2,   8, 0, 4],
[0, 0, 0,   0, 0, 0,   0, 0, 0],
[3, 0, 0,   8, 0, 0,   0, 0, 5]
```

Figure 1: Sudoku representation

From now we refer to peer of a cell x as all the cells that are in the same row, column or box of x.

2 Constraint satisfaction approach

2.1 CSP and Sudoku as CSP

Constraint satisfaction problem or CSP is definied by three elements: variables, domain and constraints.
So let's identify the problem of solving a Sudoku as a CSP:

- Variables: all the cells in the grid, so we have 81 variables;
- Domains: the associated domain to each variable is any integer numbers between 1 and 9 and it represents the possible values that can be assigned to its corresponding variable;
- Constraints: are given by the game rules and are the rows constraint, the columns constraint and the boxes constraint (direct constraints).
  - rows constraint, in each row same digit has to appear exactly once;
  - columns constraint, in each column same digit has to appear exactly once;
  - box constraint, in each box (3x3 square) same digit has to appear exactly once.

We can also identify some indirect constraints: each digit must appear in each row, column, and box. This constraints will help us to reduce the domain of the empty cells by looking at the values of the filled cells.
Obviously a possible solution is to use brute force and try all the possible combinations of numbers for each cell, until we find a correct configuration that satisfies all the constraints.
With Constrain Satisfaction problem we assign to each cell a value from its domain such that all the constraints are fulfilled. CSP algorithms don't always bring to optimal solution, they can also reach some local one or find all the possible solutions. As we are going to see our implementation of the algorithm will stop after reaching a solution.

2.2 Constraint propagation and backtracking

To avoid the brute force approach that will take a lot of time, we can use Constraint Propagation (Forward Checking) to eliminate the values in the domains that won't bring the algorithm to a correct solution. We have that to each cell there is an associated domain that represent all the possible values that we can assign to it. Constraint propagation applied to the sudoku problem will remove from the domains of the cells the values that don't satisfy the previously defined constraints. Therefore, when we assign a digit d to a cell x, the algorithm with constraint propagation will remove d from all the domains of the peers of x.
Most of sudoku puzzle that can be find in the magazines can be solved only with constraint propagation step, but for the harder ones can happen that for some cells the values in the domain are more then 1 so the algorithm should make a choice, if the choice isn't the one which bring the algorithm to the solution state, it should stop and go back in the state where it made the wrong choice to try with a different value. To implement this feature we can use the Backtracking search.
Backtracking is a depth-first search in a state space tree.

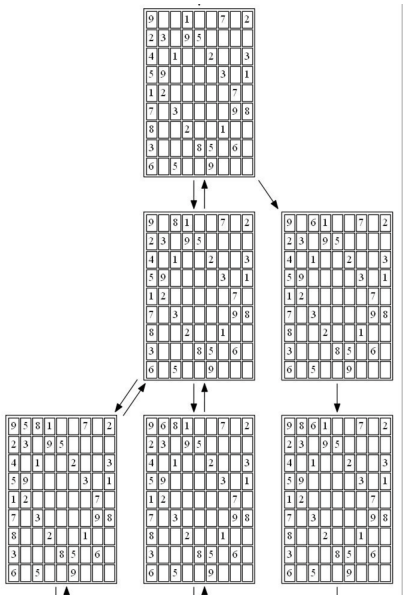

Figure 2: An example of state space tree [4]

The state space tree for the suduku is a tree where each node represents a possible state of the problem, in other words each node represents a possible assignment of a value to a cell. The subtrees of a node are all the assignment that can be made starting from a particular configuration.

So, with the backtracking we search among all the possible configurations to find one that represents the solution. But applying the backtracking search alone, the time complexity will be very high because it will explore all the branches until the solution is found. To trim the possible branches of the tree, we can combine constraint propagation algorithm with the backtracking search. In this way, with constraint propagation, we dynamically prune the state space tree as we traverse it with backtracking search.

To further reduce the number of possible attempts and to increase the probability that the algorithm will choose the correct branch, we decided to add some heuristics.
To avoid random choice of the next cell to fill, we decided to implement the Minimum Remaining Variable heuristic (MRV), so the next cell to fill with a digit will be the cell with the smallest cardinality of the domain. This heuristic should also help to discover earlier a wrong assignment.

Another heuristic that we took advantage of is the Least Constrained Value (LCV) Heuristic. When we have to fill a cell, we have to choose a value from its domain, with the use of this heuristic we try first with the digit that has less occurrences in the domains of its peers.
By doing this we have two different heuristic on two different levels that help us to reduce the branches that the backtracking search has to explore.

2.3 Implementation

Now we are going to see how we implemented the heuristics, the constraint propagation, the backtracking search and some auxiliary function.
First of all we write a function `init_domain_matrix` to create a 9x9 matrix which contain all the possible digit for each cell.

```
def init_domain_matrix():
    return [['123456789' for i in range(9)] for j in range(9)]
```

Then we have the functions `reduce_domain and propagate_reduction`, these two together implement the behavior of forward checking. In particular, `reduce_domain` is used to search for cells with an assigned value and to reduce their domain to the empty set because for sure we are not going to put another value inside of them. After reducing the domain of a cell with an assigned value, we have also to remove that value from the domains of its peers, so we call the `propagate_reduction` function.
`propagate_reduction` looks for a given digit among all the peers of a given cell, when it finds a peer's domain that contains that value, it simply removes the digit from that domain. It continues its search until all the peer's domains are checked.

```
def reduce_domain(puzzle, domains):
    for r in range(9):
        for c in range(9):
            if puzzle[r][c] != 0:
                domains[r][c] = ''
                domains = propagate_reduction(puzzle[r][c], puzzle, domains, r, c)
    return domains


def propagate_reduction(value, puzzle, domains, row, column):
    str_val = str(value)
    domains[row][column] = ''

    for i in range(9):
        tmp = domains[row][i]
        domains[row][i] = tmp.replace(str_val, '')

    for i in range(9):
        tmp = domains[i][column]
        domains[i][column] = tmp.replace(str_val, '')

    boxRow = row - row % 3
    boxCol = column - column % 3
    for i in range(3):
        for j in range(3):
            if str_val in domains[boxRow + i][boxCol + j]:
                tmp = domains[boxRow + i][boxCol + j]
                domains[boxRow + i][boxCol + j] = tmp.replace(str_val, '')
```

```
    return domains
```

`getNext2Fill` is how we implemented the MRV heuristic. It creates a list with the lengths of each domain, it searches for the minimum value in this list and then it return the row index and column index of the cell with less values in the domain. We decided to assign length 10 to the domains which are empty set. If all the domains have length 10, it means that the algorithm assigned a value to each cell.

```
def getNext2Fill(puzzle, domains):
    flat_domains = [item for sublist in domains for item in sublist]
    dom_len_list = list(map(domain_length, flat_domains))
    min_len = min(dom_len_list)
    if min_len == 10:
        return None, None
    else:
        index = dom_len_list.index(min_len)
        return index // 9, index % 9
```

`sort_domain` implements the Least Constrained Value heuristic. What we do, is sorting the domain of a given cells by the number of occurrences that a digit has in the domains of the peers of that cells.

```
    def sort_domain(row, col, dom, domains):
    numbers = []
    for i in range(9):
        temp = list(map(int, str2list(domains[row][i])))
        for j in dom:
            if j in temp:
                numbers.append(j)

    for i in range(9):
        temp = list(map(int, str2list(domains[i][col])))
```

```
        for j in dom:
            if j in temp:
                numbers.append(j)

    boxRow = row - row % 3
    boxCol = col - col % 3
    for i in range(3):
        for j in range(3):
            temp = list(map(int, str2list(domains[boxRow + i][boxCol + j])))
            for j in dom:
                if j in temp:
                    numbers.append(j)

    result = sorted(set(numbers), key = lambda ele: numbers.count(ele))
    return result
```

The core of the algorithm is the recursive function `solve_sudoku`. It basically does a recursive depth-first search, trying to put, in the cells suggested by `getNext2Fill`, the first value it finds in the sorted domain by `sort_domain`. Let see in more depth its behavior.
Firstly it gets the position (row and column) of the next cell where it is going to make a guess. Then it checks if the just obtained indexes are valid, if not it means that we don't have any other cells to fill and we are done. Otherwise, it has to sort the domain of the selected cell and for each value in the domain, it checks if the value can be a valid assignment. If it is valid, modify the value of that cell, reduce its domain and reduce the domains of its peers. At this point if the assignment hasn't generated unfilled cells with empty domain go on with the next cell. If the assignment isn't valid, the algorithm restore the value of the designed cell to '0' and restore its domain to the state before the assignment.

```
def solve_sudoku(puzzle, domains):
    row, col = getNext2Fill(puzzle, domains)

    if row is None:
        return True
    else:
        dom1 = list(map(int, str2list(domains[row][col])))
        if len(dom1) > 1:
```

```
        dom = sort_domain(row,col,dom1,domains)
    else:
        dom = copy.deepcopy(dom1)

    for guess in dom:
        if is_valid(puzzle, guess, row, col):
            puzzle[row][col] = guess
            tmp_dom = copy.deepcopy(domains)
            domains = propagate_reduction(guess, puzzle, domains, row, col)

            if error_empty_domain(puzzle, domains):
                if solve_sudoku(puzzle, domains):
                    return True backtrack e usare un nuovo numero
        puzzle[row][col] = 0
        domains = tmp_dom

    return False
```

The function `solve_sudoku` is called on a puzzle which domain is already been processed once by the function `reduce_domain`, so some invalid assignments are already removed from the domains of the cells.
The backtracking search guarantees to converge at some point, and to reduce the waste of time, we apply constraint propagation before and during the search. Moreover we used also some heuristic on variable and on values of domains to analyze first the branches with less probability to fail first, thus increasing the speed of the algorithm.

3 Relaxation labeling approach

3.1 How it works

The other approach we used to solve the problem of complete a sudoku puzzle is the Relaxation Labeling algorithm. Let's first have a brief introduction and then how we applied to our problem.
A labeling problem can be defined as:

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

The goal of these kind of problems is to assign a label of L to each object of B.
Sometimes local information isn't enough so also contextual information is exploited. Contextual information is expressed in terms of $n^2 * m^2$ matrix of compatibility coefficients: $R = \{r_{i,j}(\lambda, \mu)\}$.
Where the compatibility function $r_{i,j}(\lambda, \mu)$ measures the strength of compatibility between the two hypothesis "b_i is labeled with λ" and "b_j is labeled with μ".

Now, we can see the sudoku puzzle as a Relaxation Labeling problem, that we define as:

- the set of objects B is the set of the cells, in a 9x9 grid we have 81 cells;
- the set of labels Λ has size 9 and it is the set of digits {1,2,3,4,5,6,7,8,9}.

Therefore, we have that the compatibility function returns 1 if an assignment of a digit to a cell satisfies the game rules, else it returns 0.

The algorithm in the begin provide to each object of B a probability vector of length m:

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

With $p_i^{(0)}(\lambda) \geq 0$ and $\sum_\lambda p_i^{(0)}(\lambda) = 1$.
Each $p_i^{(0)}(\lambda)$ represents the probability distribution that the object b_i at time 0 is labeled with λ.
If we concatenate all of these probabilities vectors $p_1^{(0)}, ..., p_n^{(0)}$, we get the

initial weighted labeling assignments $p^{(0)} \in \mathbb{R}^{nm}$. The space of weighted labeling assignments is:

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

where Δ is the standard simplex of $\mathbb{R}^n$. Each vertex of IK represents an unambiguous labeling assignment.
The optimal solution of a labeling problem is to reach a point where there are only unambiguous labeling assignment, that means: for each cell i of the sudoku, one digit out of 9 has the highest probability to be assigned to cell i.
On the other hand, we have an ambiguous assignment when for example for the cell $(0, 0)$ we have a probability distribution like this $p_0^{(t)} = (0, 0.2, 0, 0, 0.4, 0, 0, 0.4, 0)$, that means digits 5 and 8 have the same probability to be chosen for the assignment to that cell.
The relaxation labeling algorithm starts with a weighted labeling assignment as input and iteratively updates it considering the coefficients in the matrix R. Unfortunately, the algorithm doesn't always converge.
And if it does, it may not end with only unambiguous labeling assignment for each object that meets all the rules of the game. So it can end with probability vectors like the one in the above example, or it can end assigning a value to a cell which doesn't satisfy the game constraints.

The probability vectors are updated according to the following formulas introduced heuristically 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,

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

and $q_i^{(t)}(\lambda)$ quantifies the support that context gives at time t to the hypothesis "b_i is labeled with λ" at time t.

The stopping criteria we used to decide when to stop the algorithm is to compute the euclidean distance between each probability vector at times $t + 1$ and at t, and stop when it is smaller than a given threshold.

3.2 Implementation

To make the relaxation labeling at least comparable with the other approach we needed to use matrices to store the compatibility coefficients, the qs and the probabilities p. The first function we implemented is `initBoard`, it creates a vector of length 729, where we have a position for each possible label of each sudoku cell. Basically we store sequentially the probability vectors of each object. For the cells with an assigned value, the probability vector will be set to zero but in the position of the assigned digit there is a 1. For the empty cell, with the help of the direct constraints, we set to 0 the probabilities of the labels that can't be assigned, and the remaining probabilities to a random number between $\frac{1}{n} - 0.005$ and $\frac{1}{n} + 0.005$, where n is the number of remaining possible labels to be assigned to that cell. We needed to add a random oscillation from the above interval in order to reduce the probability of ending with probability vectors in which two or more labels have the same probability.

```
def initBoard(board):
    p = np.ones((TOT_OBJECTS * SIDE, 1)) / SIDE

    for i in range(SIDE):
        for j in range(SIDE):
            domain_set = getPossibleValue(i, j, board)
            n = len(domain_set)
            prob = np.zeros((1, SIDE))[0]
            if board[i][j] != 0:
                val = int(board[i][j])
                prob[val - 1] = 1
            else:
                for k in domain_set:
                    prob[int(k) - 1] = np.random.uniform(1 / n - 0.005, 1 / n + 0.005)
            prob = prob / np.sum(prob)
            p.reshape(SIDE, SIDE, SIDE)[i][j] = prob
    return p
```

As said before we decided to save all the compatibility coefficients in a matrix to speedup the algorithm. The following function `R_matrix` does exactly this, for each possible pair of labels (λ, μ) of each possible pair of cells $(i,\ j)$, we compute the compatibility function $r_{i,j}(\lambda, \mu)$ and then we assign the returned value to the relative cell of the matrix.

```
def R_matrix():
    rij = np.zeros((TOT_OBJECTS * SIDE, TOT_OBJECTS * SIDE))
    for i in range(TOT_OBJECTS):
        for lmbda in range(SIDE):
            for j in range(TOT_OBJECTS):
                for mu in range(SIDE):
                    rij[i * SIDE + lmbda][j * SIDE + mu] = function_r(i, j, lmbda, mu)
    return rij
```

`relaxation_labeling` takes in input the vector of probability distributions. Firstly it creates the matrix of compatibility coefficients then it iterates until the euclidean distance between the previous and the updated vector of probability distributions is less then 0.001. With the assignment `q = np.dot(rij, p)` we create a vector where we store sequentially all the q for each label of each object at a particular time. Then with fourth assignment in the while statement, we update the probability $p_i^{(t+1)}(\lambda)$ for each label $\lambda \in \Lambda$ and for each object $b_i \in B$.

```
def relaxation_labeling(p):
    rij = R_matrix()
    prev = 0
    diff = 1
    step = 0
    while diff > 0.001:
        q = np.dot(rij, p)
        numeratore = p * q
        denominatore = numeratore.reshape(TOT_OBJECTS, SIDE).sum(axis=1)
        p = (numeratore.reshape(TOT_OBJECTS, SIDE) / denominatore[:, np.newaxis]).reshape(TOT_OB

        diff = np.linalg.norm(p-prev)
        print("Euclidian distance: ", diff, ", step: ", step, "")
```

```
        prev = p
        step += 1
    return p
```

The euclidean distance is an heuristic that for the most of the cases allow to reach a point where the algorithm should converge, not too early, and each probability distribution is changed enough to correctly indicate which label should be assigned to the according cell.

When the relaxation labeling algorithm converge, in the function `apply_sudoku` we look at the label with the highest probability in each probability vector, and we assign that label to the according cell.

```
def apply_sudoku(sudoku):
    p = relaxation_labelling(initBoard(sudoku))

    for i in range(SIDE * SIDE):
        pos = np.argmax(p.reshape(TOT_OBJECTS, SIDE)[i])
        sudoku[i // SIDE][i % SIDE] = pos + 1
    return sudoku
```

4 Comparison and Conclusions

The **constraint satisfaction** approach, based on constraint propagation and backtracking guarantees convergence, because for any sudoku the number of possible state is finite so at a certain time the backtracking search will have explored them all. The drawback is if the sudoku is hard to solve, the number of branches will increase drastically, then the search will take a lot of time to find the right solution. But with the help of constraint propagation and the two heuristics we described before, we can decrease the number of branches so the backtracking will have less states to check.
Instead, the **relaxation labeling** approach even with the use of the heuristic, can't guarantee convergence and it can solve only easy puzzles. In fact with the increase of the complexity of the sudoku, it can happen that at the end of the algorithm two or more label have high and similar probabilities. In these cases there are no constraint to use to choose the right digit to assign, so the algorithm could return wrong solutions or take a very very long time to end.

To have more metrics to compare the two approaches, we measured the time that each approach needed to solve four different puzzles 100 times each.

For Constraint satisfaction approach,

- `board_p`: 0.0432 sec;
- `board1`: 0.0448 sec;
- `board2`: 0.0522 sec;
- `board_n`: 0.0569 sec.

For Relaxation Labeling approach,

- `board_p`: 27.1077 sec;
- `board1`: 26.7595 sec;
- `board2`: 25.6091 sec;
- `board_n`: 25.3200 sec.

As we can see for the sudokus that we tried, the first approach is in average about 500 times faster than the second one. The 4 puzzles are in increasing order of complexity. In particular we noticed that for `board2` and `board_n`, the first algorithm needed to do several steps of backtracking however the time is increased only by 0.01 sec and it converged to the right solutions. While, the second algorithm even if it needed less time to converge, it returned wrong solutions for both puzzles.

Let be $m = 81$ and $n = 9$, we now compare the temporal complexity and spatial complexity of both oh them.

- Temporal complexity: for CSP with backtracking approach we have $O(n^m)$, all the possible states generated by a grid 9*9. Luckily we can limit all these branches with the help of constraint propagation and the MRV and LCV heuristics. For RL approach we don't really know how much it will take to end because of the use of the heuristic based on euclidean distance. Neither we know if the RL algorithm will ever converge in a finite time.

- Spatial complexity: for CSP with backtracking approach, we have $O(n* m)$, the dimension m of the grid that for each cell stores a domain of length at most n. For RL we also have $O(n * m)$, for the matrices of p and q, but to store all the compatibility coefficients we used a matrix that takes $O(n^2 * m^2)$.

So, we can see that the first approach is better for both complexities.

To conclude, constraint satisfaction approach based on constraint propagation and backtracking, always converges to a correct solution, if it exists, even with more complex puzzles, it only needs more time. RL can solve only simple sudoku, otherwise it can end with a wrong solution or it can take forever to converge. What guarantees the convergence of the backtracking is the fact that it is based on the DFS search, while RL uses an heuristic. To reduce the time of each iteration of the RL, we needed to vectorize almost everything we used, otherwise it takes a lot of time, but this increased a lot its spatial complexity.
By this comparison we can say that the approach based on backtracking and constraint propagation can work very well on both easy and hard puzzles and it can be easily improved with the help of some heuristics. Note that we implemented the constraint propagation only with the forward checking, probably if we did it with arc consistency, we could achieve even better performances.
On the other hand, the relaxation labeling algorithm can solve only simple sudokus otherwise convergence and correctness can't be guarantee. So maybe this approach isn't to appropriate really solve this kind of problems. Probably, in context where contextual information has a bigger impact, we could have very different results and RL could perform way better.

References

[1] Sudoku as CSP: https://www.codeproject.com/Articles/34403/Sudoku-asaCSP

[2] Implementing a CSP solver for Sudoku: https://dfsxpertsys.com/downloads/ARAISudokuCSPBittner-Oosting2011.pdf

[3] Suduku puzzles via relaxation labeling: https://www.cs.bgu.ac.il/~benshahar/Teaching/Computational-Vision/StudentProjects/ICBV071/ICBV20071-VladimirGoldman/index.php

[4] Solving Sudoku with CSP and Backtracking https://sandipanweb.wordpress.com/2017/03/17/solving-sudoku-as-a-constraint-satisfaction-problem-using-constraint-propagation-with-arc-consistency-checking-and-then-backtracking-with-minimum-remaning-value-heuristic-and-forward-checking/