Università
Ca'Foscari
Venezia

REPORT OF SUDOKU SOLVER

Luca Bizzotto

875814

Anno accademico 2021/2022

Indice

0.1 Introduction

The problem addressed is to implement a Sudoku solver using a constraint satisfaction approach based on constraint propagation and backtracking and one based on Relaxation Labeling.
Then we'll provide an analyses doing a comparison between those two approaches, their strengths and weaknesses.

0.2 Sudoku

The standard Sudoku puzzle consists of a 9 x 9 grid, broken into nine 3 x 3 boxes.
The goal of this game is to assign to each square a digit from 1 to 9 such that for each block, each row and column must contain the numbers from 1 to 9, without repetitions.
The Sudoku is given with some filled square, the player then has to complete the rest of squares such that the rules are satisfied.
A Sudoku puzzle can generates 4 x 10^{38}possible states, so a brute force approach to solve it is not efficient in terms of time and space.
It is necessary to reduce this large space without searching in states for which is not possible to reach the solution.
One basic solution consists of taking advantage from the constraints so we don't have to search for all 4 x 10^{38} states because when we explore a new state we can immediately eliminate many other states which are not consistent with game rules. Following this idea we can say that Sudoku can be seen as a particular problem in which game rules limit the search space, allowing to develop efficient algorithms for solving it.

0.3 Constraint propagation and backtracking

Constraint propagation is an elementary method for reducing the search space of combinatorial search and optimization problems.
The basic idea of constraint propagation is to detect and remove inconsistent variable assignments that cannot participate in any feasible solution.
For example in sudoku game the algorithm removes the values that will not satisfy constraint properties of the game.

There are situation where using constraint propagation does not guarantee a solution, in general, when we have to many values in domain and/or constraints are weak.

Most of the Sudoku games can be solved easily with simple constraints propagation, bringing to the final solution very fast, but for some particular configurations a correct result is not guarantee. For this reason combination with another technique called backtracking it is a good choice to implement an efficient algorithm.

The backtracking technique allow us to explore all the possible solution, since it does a complete search of the state space tree using a depth search.

A state space tree it a tree in which each node represent a possible state of the problem. If at any step of the algorithm it becomes clear that a solution in the current path is not possible then the algorithm goes back to a consistent state and search in another branch.

Using pure backtracking however has some drawbacks, in fact pure backtracking explore all possible solution without making any pruning among all possible choices, and it also has to keep in memory each state since in case of backtrack it has to restore it in stable state.

So for those reasons it is good idea a combinations of backtracking with constraint propagation in this way we reduce the number of explored space by pruning inconsistent branch and the drawbacks of the pure backtracking are reduced.

0.4 Approach

Sudoku problem for its rules definition can be modelled as a Constraint Satisfaction Problem.
The variables are represented by each cell on the game board (81variables) and the initial domain set of each variable is composed by the range of values starting from 1 to 9.
Constraints are given by game rules and can be subdivided in three categories:

- Row constraint: values on the same row have to be distinct.
- Column constraint : values in the same column have to be distinct
- Box constraint: values in the same block (3 x 3 square) have to be distinct

In the following we describe the overall idea of the algorithm.
The first step is to find a cell that is still unsigned, if no one cell is founded this means that all the cells are properly assigned and the Sudoku is completely fill up. Otherwise if an empty cell is founded we proceed with its initialization in respect of the constraint rules imposed by the game. To represent a cell we used an object that contain the cell's domain and cell coordinate respect the Sudoku grid. Then the following step is to make a try for the cell that we previously picked, so one of the possible values of the domains of the cell is select and insert in the Sudoku matrix in its correct position, and then we propagate the result using recursion, in particular using a depth search.
Off course since we are making a guess for each cell from a list of possible values there is not certainty that the guessed value is right,meaning that it probably can bring to an unsolvable state.
If at any point the recursion process gives us a negative result then we have to activate the backtracking that recover the states.
Thank to the power of recursion when we are coming back from the call in case

of wrong guess,it is possible to go back in a stable state and searching for another guess(we try another possible value of the domain if still possible). The recover to a stable state it result quite easy thank to the stack. This behaviour in fact simulate what we describe before as backtracking algorithm.
Since the recursion is called inside a while cycle at each step it tried a new possible value of the reaming domain of the cell.
So if at any point the domain of the cell is equal to 0 and not all the cell of the Sudoku matrix are assigned mean that is not a possible valid configuration and we have to return a false to the caller.
In the following will be represent the pseucode for Constraint satisfaction problem with backtracking:

$INPUT \leftarrow Sudoku$

$OUTPUT \leftarrow Solved\_SudokuM$

$solve() \leftarrow cordinate$ ▷ main function

1: $solve()$
2: $cell \leftarrow find\_unsigned\_cell()$
3: **if** $!cell$ **then**
4: $return \leftarrow true$
5: **else**
6: $cell.update\_domain()$
7: **while** $cell.domain() \neq \emptyset$ **do**
8: $Sudoku[cell.coordinate] \leftarrow choose(domain.value())$
9: **if** $solve()$ **then**
10: $return \leftarrow true$
11: **end if**
12: **end while**
13: $Sudoku[cell.coordinate] \leftarrow BLANK$
14: $return \leftarrow false$
15: **end if**

In general backtracking uses fixed ordering of variables, but this is not always efficient.
On the repository of the project you can find another file called Csp_rnd.h in which we try to use the variables in non a fixed ordering.
We know that sometimes in some hard problem when the introduction a stochastic choice the performance or the result can be improve. So our idea was to introduce a stochastic choice to choose the next cell to try to assigned among all possible unsigned cell.
In this paragraph we just specified a the global idea that we used for our solution, more implementation details are explained in the code comments.

0.5 Relaxation Labeling Algorithm

Relaxation labeling it is an approach that consist of not use only local information but also consider contextual information.
A labeling problem is defined by :

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

The goal of this technique is to label each object of B with a label of Λ.
Contextual information are expressed in terms of real-valued $n^2 \times m^2$ matrix of compatibility coefficients $R = \{r_{ij}(\lambda, \mu)\}$.
The coefficient $r_{ij}(\lambda, \mu)$ measures the strength of compatibility between the two hypotheses: "b_i is labeled λ" and "b_j is labeled μ".
A relaxation labeling algorithm start with an initial m-dimensional probability vector for each object $i \in B$

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

with $p_i^{(0)}(\lambda) \geq 0$ and $\sum_\lambda p_i^0(\lambda) = 1$.
Each $p_i^{(0)}(\lambda)$ represents the initial, non contextual degree of confidence in the hypothesis "b is labeled λ. Each object is associated to one probability distribution, and the concatenation of all of these defines a weighted labeling assignment $p^{(0)} \in \mathbb{R}^{nm}$.All the possible weighted labeling assignment belong into a space IK is:

$$\text{IK}= \Delta^m = \Delta \times ... \times \Delta$$

Where each Δ is the standard simlex o $\mathbb{R}^n$. Each vertex of IK represents an unambiguous labeling assignment, that is one which assigns exactly one label to each object.
A relaxation labelling process takes the initial labeling assignment p(0) as input and then iteratively reduce the ambiguity and disagreement of the initial

labels considering the compatibility r_{ij} previous defined. The idea is that at the end of the iterative procedure all the probability distributions have an high probability, ideally equal to 1, to the right label that should be positioned in that cell. Relaxation strategies do not necessarily guarantee convergence, and thus, we may not arrive at a final labelling solution with a unique label having probability one for each feature. In fact, at the end of the iterative procedure it can happen that some vectors provide ambiguity or inconsistent assignments. Each iteration step updates the vectors probability using the following heuristic formula, provided by Rosenfeld, Hummel and Zucker in 1976:

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

Where

$$q_i^{(t)}(\lambda) = \sum_j \sum_\mu r_{ij}(\lambda, \mu) p_i^{(t)(\mu)} \tag{2}$$

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

Averages local consistency evaluates the stopping criteria of the algorithm

$$A(p) = \sum_i \sum_\lambda p_i(\lambda) q_i(\lambda) \tag{3}$$

The stopping criteria has to consider the convergence of the algorithm.

When the step measure is lower then a given threshold " the algorithm stops, meaning that it has reached the convergence.

0.6 Approach

Before to introduce the implementing idea of the algorithm we define the Object B and the Set of labels Λ for this problem:

$$B = \{cell_1, ..., cell_{81}\}\ \Lambda = \{1, ..., 9\}$$

We choose to implement the matrix as a map with key equal to the coordinate of the object b_i respect to Sudoku grid and value equal to a vector V of size 9. Each value of the vector V represents the probability that the label with value Λ be associated with that particular object.
Instead of going to patch all the cells of our matrix we decided not to insert the cells with values already defined in this way we can create a matrix saving some space therefore B is the set of blank cell and Λ is the set of labels.
So we can obtain the P matrix a time 0 and with equation 2 we can obtain matrix Q that holds all $b_i \in B$ and all λ in Λ. To define the probability vector for each cell we decided to go first to determine the possible values that that particular cell could have and then we calculated the probability of each possible label using a uniform distribution:

$$\forall\ \lambda \in \Lambda_{b_i}\ p_{\lambda^0} = \frac{1}{|\Lambda_{b_i}|}$$

To calculate the coefficient $r\_ij(\lambda, \mu)$ that measures the strength of compatibility between the two hypotheses: "b_i is labeled λ" and "b_j is labeled μ" we used the following algorithm:

1: $INPUT \leftarrow b\_i, b\_j, \lambda, \mu$
2: **if** $b\_i == b\_j$ **then**
3: $\quad return \leftarrow false$
4: **end if**
5: **if** $\lambda \neq \mu$ **then**
6: $\quad return \leftarrow true$
7: **end if**
8: **if** $same\_row()||same\_column()||same\_square()$ **then**
9: $\quad return \leftarrow false$
10: **else**
11: $\quad return \leftarrow true$
12: **end if**

To compute the algorithm now we have to use the equation 1 to build the matrix P a time $t + 1$ and the equation 2 to build the matrix Q a time $t + 1$. Then we keep going to iterate until the difference from the average consistency illustrated in equation 3 of $P^{(t)}$ is close to the average consistency of $P^{(t+1)}$. Below the pseducodice :

1: $solve()$
2: **while** $diff \leq 0.01$ **do**
3: $\quad diff \leftarrow average\_consistency()$
4: $\quad calcolate\_P(t + 1)$
5: $\quad calcolate\_Q(t + 1)$
6: $\quad P^t \leftarrow P^{t+1}$
7: $\quad diff \leftarrow abs(diff - average\_consistency())$
8: **end while**

At the end of the two procedures it is necessary to iterate for each blank cell of the Sudoku puzzle and fill the cell with the label that has the larger probability inside the distribution.

0.7 Performance

To analyze the performance we decided to use a data-set of Kaggle. Each instance of the data-set is also represented by a difficulty level. For each Sudoku taken into consideration the level of difficulty goes from 0 to 8. In order to test the efficiency and efficacy of the two technique 4 levels of complexity of the Sudoku are considered:

- Easy with difficulty level equal to 0.0
- Medium with difficulty from 0.1 up to 2.7
- Hard with difficulty from 2.8 up to 5.4
- Expert with difficulty from 5.5 up to to 8.0

The measures that we choose to consider in this analysis are the following one:

- Sudoku rating
- Correctness
- Number of iteration
- Number of backtracking invoked for CSP

We choose to not include time because it also depends on the machine in which the algorithm is run.
On the following we find a table that reports the average of the values compared to the tests performed with the back constraint propagation technique.

The table 1 summarizes the behavior of the algorithm to vary the difficulty of Sudoku. We can therefore note that as the difficulty increases, the number of times backtracking is activated and consequently the number of nodes visited increases. We can also observe that the solution with this technique is always guaranteed even with more complex Sudoku, but with the cost of greater complexity.

Difficulty	Visited nodes	Backtracking	Solved
Easy	137	88	true
Medium	131930	131873	true
Hard	162438	162380	true
Expert	165724	165667	true

TABELLA 1: CSP algorithm results

Difficulty	Visited nodes	Backtracking	Solved
Easy	137	88	true
Medium	131930	131873	true

TABELLA 2: Rl algorithm results

The table 2 summarizes the results obtained as the complexity of Sudoku varies. For the analysis of this technique we have grouped the classes of complexity of Sudoku in easy with complexity equal to 0 and hard the contain all the remain possible values of complexity as it manages to solve only Sudoku of easy complexity.

The limit used by us as the stop criterion is 0.01. We can note that with the increase of the complexity of Sudoku taken into consideration also increases the number of iterations made, moreover with this technique the correct resolution is not guaranteed, in fact we can see that only Sudoku characterized by easy complexity are solved in the correct way.

The table number 3 goes instead to describe the performance of the disease that is based on the use of a stochastic method to choose the next cell to be assigned. The solution in this case is always guaranteed for any type of Sudoku level , from the point of view of performance instead is highly lacking compared to the algorithm that chooses the variables in fixed order.

As we can see the number of nodes visited is highly higher than its corripsective in the classic method. We can however say that in this case a stochastic method in the choice of the next cell to be assigned is not performing

Difficulty	Visited nodes	Backtracking	Solved
Easy	106474071	106474015	true

TABELLA 3: CSP random algorithm results

0.8 Conclusion

In conclusion we then applied two different techniques the CSP and RL to solve the problem of Sudoku. We also performed an analysis going to test our algorithms with different instances of play and with different levels of difficulty. Let's now analyze in detail the following fields:

- Completeness :
  - CPB :As far as the contribution to the results obtained, we can say that the completeness it is guaranteed because if the solution exists it is always found
  - RL: No, it solves only easy Sudoku.
- Optimality:
  - CPB: In general there is no optimal solution as there are may be several solutions and all are correct
  - RL: no, since it solves only easy Sudoku
- Temporal complexity:
  - CPB: $O(m \times n)$. we say this by doing an overestimation in which we know that using the constraint propagation are carried out pruning and therefore we will never analyze all the tree of solutions
  - RL: we know that each updating step cost $\Theta(n^2 \times m^2)$ but we cannot define an upper bound since we will continue to iterate until the stopping criteria based on a heuristic will be reached

- Spatial complexity:
  - CPB: considering that we have taken into account only the blank cells an not all the cells of the Sudoku grid, the spatial complexity will be equal to the blank cells $\times$ domain so un upper bound it is $O(n \times m)$
  - RL: Having decided to store the matrices p and q to speed up operations instead of apply directly the formula 2 the spatial complexity will be equal to $O(n^2 \times m^2)$