Artificial Intelligence: Assignment 1

March 11, 2016

Prof. Andrea Torsello

Francesco Pelosin

Sudoku with Backtracking and Constraints Propagation

We can see the Sudoku problem as a constrain satisfaction problem and we can formalize it as follows.
Formalization We can formulize the problem by defining:

- Variables/objects: represented by the 81 cells of the puzzle.
  - Variables have a domain of possible values/labels, in our case 1 to 9.
- Constraints : which are the "rule" of the game
  - No two equal numbers in the same row
  - No two equal numbers in the same column
  - No two equal numbers in the same box

Each assignment of a variable to one of its possible values, represent a state of the problem that's why we can translate the problem into a graph problem.

Graph representation

Each node assignment represent a particular Sudoku state. It is easy to see that if we consider one cell at each step, we have a branching factor of nine possible assignments for a given cell and in particular, the whole graph, has 9^81 nodes. Obviously, the majority of these nodes are failing nodes or descendant of them, so they are irrelevant for the solution. A given Sudoku puzzle has some initial assignments, so the algorithm will start in an advanced state. The solver algorithm should find the best assignments to search the goal state, which is the state where all the cells have been assigned to a value and all of them respect all the constraints of the problem.

Plain Backtracking Algorithm

Idea

The simple backtracking algorithm is a Depth-First Search in the graph, in the sense that it will try to find a solution by trying every possible value in the domain of a given cell, and if it find a feasible value, it will make the assignment and move deeper into the graph by repeating the process. If it encounter a cell with an empty domain (failing node), then it return to the previous cell and try with the remaining values.

Evaluation

Completeness Does the algorithm find a solution? The answer is yes, because it will eventually try every single possible configuration to solve the Sudoku. This could result in a huge temporal complexity but it will surely find a solution, at least in our Sudoku problem, which does not have any infinite-deep branches where the algorithm could stuck in.

Optimality Will the algorithm find the "best" solution when more than one exist? Given the fact that in Sudoku there could be different solutions to a given puzzle, and if we interpret best solution" with the "nearest solution from the initial state"', the answer is no. The problem here is that the algorithm does not search with a heuristic, it just try to go deeper and deeper until it will find a solution or it cannot go any further, then it will consider another path. This does not guarantee to find the best solution; it is just a matter of "lucky" choice of the branches.

Temporal Complexity As we said, the simple backtracking algorithm could try all every possible configuration for a given puzzle, resulting in a huge temporal complexity. In the worst case, depth-first search

```
Require: Nothing apart the matrix of cells
function BACKTRACKING(sudoku cells)
    for all cells in the matrix do
        if cell is empty then
            for all possible labels do
                current_cell = label
                if the assignment respect all the constrains then
                    ris = BACKTRACKING(sudoku cells)
                    if ris then
                        return true
                    else
                        continue                          ▷ We try other labels
            current_cell = 0
            return false
    return true
```

Algorithm 1: Simple Backtracking

will iterate through all of the $O(9^m)$ nodes in the search tree, where 9 is the branching factor (possible values for the cell i.e. domain cardinality) and m is the maximum depth of the search graph (which depends from the initial state i.e. the initial assignment of the given puzzle), which is still a great amount of time.

Spatial Complexity Spatial complexity is not a big deal, the backtracking algorithm just stores one instance of the puzzle and modifies it every time it computes the assignment or rollback the assignment after detecting a fail node, it will take in main memory simply a matrix of 9×9 integers, which is actually small for this particular problem.

Considerations

Thanks to the great power of nowadays CPUs the answer is almost instant, but it remains a "stupid" solution because the problem with this algorithm is that it is somehow greedy, in the sense that it does not "know" what it is doing, and it does not know how its actions modify the context (near cells), in fact the latter consideration is one of the features that can improve this solution. A simple but effective temporal complexity improvement could be achieved if we pass at each recursion call the current cell, this could reduce a lot of unusefull iterations.

Backtracking with Constraints Propagation Algorithm

Since the simple Backtracking solution is not so effective and does not take in consideration the "context" implications of an assignment, we might introduce some heuristic to take it in account, maybe by propagating the consequences of a choice. After that, we can introduce some other improvements in how to choose the next cell to which perform the assignment, which corresponds in some heuristics in where to look first in the search graph.

Idea

A good way to improve the plain backtracking solution is to answer the following questions:

- *What are the implications of the current variable assignments for the other unassigned variables?*

  We introduce what is called **Forward Checking**, at each time that we assign a value to a cell, we exclude from the domains of the other cells of the same row, column and box, that particular value. With this trick we propagate our decision and we reduce the values that the algorithm has to try before finding a value that respect all the constrains. This technique introduce the notion of "context information" and "contextual awareness" of the choices made. It reduces the temporal complexity by pruning some branches of the search graph that will obviously fail due to an inconsistent assignment.

- *Which cell should be assigned next, and in what order should its values be tried?*

  For the second question, we could keep a list where we count the least frequent used numbers in the current solution and we could try the least-used, first. I have not used this approach in the solution because it is a little bit pedantic, but it is a plausible approach which might give some benefits. For the first question, the answer is **Minimum Remaining Values**. We look through the cells and at each step, we choose the variable that has the lowest number of feasible labels in the domain. This obviously works if we apply forward checking that previously remove all the inconsistent values. By using this technique, the algorithm takes the node in the search graph that it is most likely to cause a failure soon.

I have noticed by using minimum remaining values as a candidate selection, that for many Sudoku puzzles at each step there is just one cell, with just one feasible value in the domain. In that case the algorithm will converge in no more than number-of-empty-cells steps and, obviously, it will be the best case, this is also the lower bound of the problem itself i.e. no algorithm can do better. Typically, it correspond to a well-formed intermediate Sudoku for humans (because it has a deterministic candidate at each step, otherwise it could be very difficult for a medium human). In any case, with this improvement, the average complexity drop drops down of several orders of magnitude.

Require: Initialization of the *mrvList* and some preliminary check of the domains

```
1:  function BacktrackingConstrains
2:      if mrvList is empty then
3:          return true
4:      else
5:          candidate = mrvList.first
6:      if candidate has no feasible labels then
7:          return false
8:      for all labels of candidate do
9:          if current_label is feasible then
10:             candidate.value = current_label
11:             RestrictDomainsConnectedTo(candidate)
12:             mrvList.sort
13:             ris = BacktrackingConstrains
14:             if ris then
15:                 return true
16:             else                                          ▷ Rollback
17:                 DerestrictDomainsConnectedTo(candidate)
18:                 candidate.value = 0
19:                 mrvList.add(candidate)
20:                 mrvList.sort
21:     return false
```

Algorithm 2: Backtracking with constrains propagation

Evaluation

Completeness The algorithm will eventually converge to a solution, so it is complete. That is because it is just a smarter version of the plain backtracking, with some tricks that allows us to prune some obviously failing branches.

Optimality In some cases, where there is always only one candidate with a single feasible value in the domain at each step (Sudoku is well formed), this algorithm is optimal with respect to the problem, but with puzzles that do not provide that formulation, it will not always converge to the optimal solution. Then we can say that in the general case it is not optimal.

Temporal Complexity The temporal complexity is much lower than the simple backtracking. For each empty cell we do not have any more a branching factor of nine because, thanks to the forward checking, we prune the failing branches, so at each level the branching factor depends from the past assignments.
When the Sudoku is well formed, the algorithm stops in $O(c)$ steps, where c is the number of the starting empty cells, as we said this is the lower bound under which it is impossible to do better.
The worst case (when the Sudoku is not well formed) it's difficult to formalize, but intuitively happens when the algorithm at each steps has to guess between several feasible labels of the candidate variable and if the algorithm always make the wrong guess, it could touch all the failing branch before choosing the right path, but this is very unusual since we use forward checking and minimum remaining value first techniques.
To partially overcome this problem, we could develop some kind of heuristic to choose the next label to try. A solution could be the least assigned label in the current step, but this improvement would require other contextual information so more spatial complexity and some more managing of the structure would be required. I have not implemented this improvement because I find it a little bit overkill for the problem, but for more complex problems it could be a valid solution.
We can say that the worst case has a complexity which is still strictly lower than the plain backtracking case: $O(9^m)$.
We improved the solution but nothing comes for free, since we introduced some "contextual" information, then we need to store them somewhere. To use minimum remaining values, in fact, we need a structure that keeps track of the cells' domains, as an ordered list. We need to sort it each time we select the next candidate or we rollback from a wrong candidate choice (i.e. we return from a failing branch), so we are introducing temporal complexity too, which depends from the sorting algorithm and the number of remaining unassigned variables at each step.

Spatial Complexity As we said, we need a support structure to implement the minimum remaining values, at the beginning it must hold all the 81 cells, then it eventually shrinks as the algorithm performs new steps. Since I wrote the program in C# the structure I choose is a `List<Cell>` and it contains the references to the cells of the puzzle. This is a simple structure, which does not require big amount of memory. To keep track of each cell's domain I introduced an array of nine Boolean values, whose respective entries are true if the value can be assigned and false if they does not respect the constrains. These new structures does not require a big amount of memory because we are adding $9 \cdot 81$ cells of Boolean values for the domains vectors and a simple list which we need to maintain ordered and whose size depends at each step.

Considerations

After running the algorithm with the Professor's Sudoku puzzle the algorithm stops in 46 steps, which is the number of the starting zero cells of the puzzle, so the algorithm is optimal. For the other Sudoku the algorithm gives an almost instant answer, but I think this is pretty natural since this method seems to me very deterministic-based and relies on a good a-priori knowledge of the problem. I would say that this algorithm is optimal towards the complexity of the problem but only if the Sudoku is well posed i.e. if it

admits only one candidate at each step.

Relaxation Labelling Algorithm

As second method to solve the Sudoku puzzle, we used a relaxation labelling algorithm, that is a method that aim to iteratively assign labels to objects by using compatibility relationships. In our case, the compatibility relationships are guided by the Sudoku constrains, and the labels and objects are conceptually the same as we discussed for the Backtracking algorithm. The core idea behind this algorithm is that at each step there is an update function that calculates the probability that a particular label should be assigned to a given cell. What makes this very interesting is that it does so only by using context information i.e. by looking at all the other variables. I've broken the algorithm in different parts, each of which has a specific functionality.

Require: Two cells: $Cell_i$, $Cell_j$ and its respective labels λ, μ
function COMPATIBILITY(λ_i, μ_j)
  if λ_i *and* μ_j *are not in the same row, column or box OR they have different label* **then**
    return true
  else
    return false

Require: A $Cell_i$ and its label λ
function SUPPORT(λ_i)
  for each $Cell_j$ **do**
    for each label μ **do**
      $ans = ans + \text{R}(\lambda_i, \mu_j) \cdot p_j(\mu)$
  return ans

Require: Nothing apart the matrix of cells
function UPDATEPROBABILITIES
  for each $Cell_i$ **do**
    for each label λ **do**
      $p_i(\lambda) = \frac{p_i(\lambda) \cdot \text{SUPPORT}(Cell_i, \lambda)}{\text{DENOMINATOR}(Cell_i)}$

function DENOMINATOR($Cell_i$)
  for each label μ **do**
    $ans = ans + p_i(\mu) \cdot \text{SUPPORT}(\mu_i)$
  return ans

Require: Nothing apart the matrix of cells
function RELAXATIONLABELLING
  repeat
    UPDATEPROBABILITIES
  until *one of the termination conditions is satisfied*
  SETCANDIDATES ▷ It will set the most probable label for each cell

Algorithm 3: Relaxation Labeling Algorithm

The algorithm proceeds by updating the probability vector of each cell. But when does it stop? I have used three type of stop condition (update tresholds) for the algorithm:

- Local Average Consistency$^{(t)}$ − Local Average Consistency$^{(t+1)} < \epsilon$
  Where the Local Average Consistency is a well known stop condition, with this termination it basically correspond to the analogous optimization problem: "maximization of the objective function within a treshold", I've set $\epsilon = 0.001$ which seems to be a good choice.

- Euclidean distance between $p_{i...n}^{(t)}$ and $p_{i...n}^{(t+1)} < \epsilon$
  Which correspond to the analogous optimization problem: find "argmax" condition. Even here I've set $\epsilon = 0.0001$ which seems to be a good choice too.

- "My termination": $p_i^{(t)} - p_i^{(t+1)} < \epsilon$
  The update of each probability vector does not variate more than a given treshold ϵ, it is a similar "argmax" termination condition. In fact seems to me that this is some kind of distance measurement, like a variation of the second termination method. I have set it to be 0.001 which seems to be a good value. This is definetively the fastest method since it has to do less computations.

I've collected some data about the variation of the Average Local Consistency and the Euclidean distance, between a state and the post-update state. Data comes from 50 different sudoku and they show the trend of the termination function before reaching the update treshold. Since I found the sudoku data set online I don't know if they are well defined or not (and I don't know if the source is trustable), but there are 13 convergent sudoku and 37 non convergent sudoku. Obviously I've set a maximum of 9000 iteration for the algorithm both for Euclidean version and ALC version, this allows us to prevent some infinite loops on non convergent sudoku, but this limitation seems to be ineffective since che algorithms always terminate before reaching it.

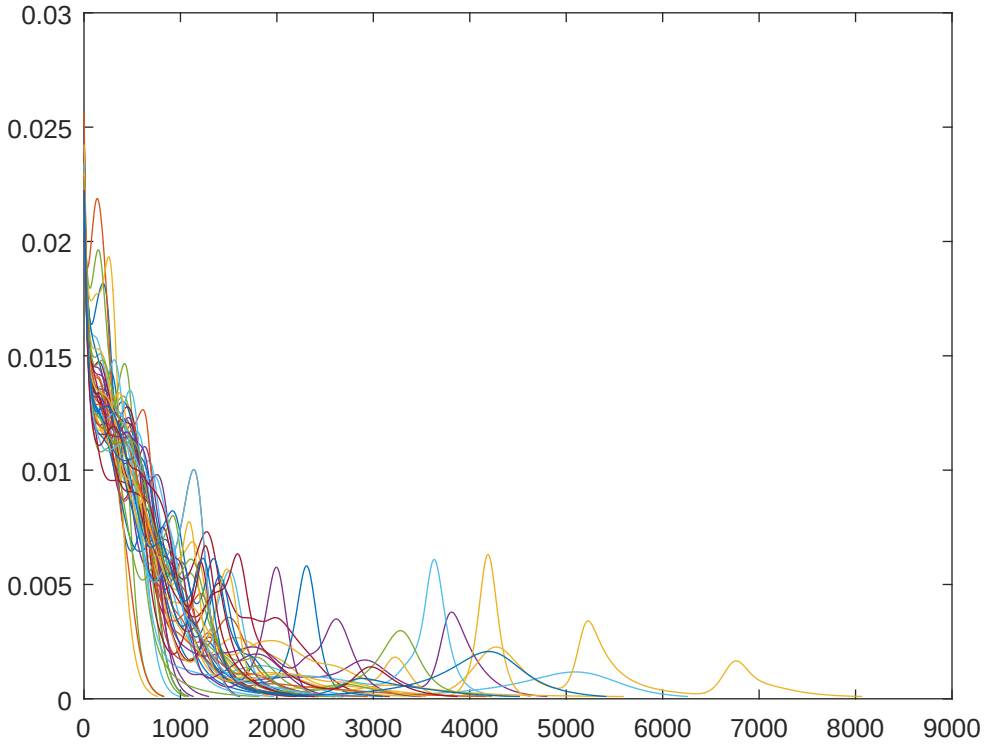

Figure 1: 50 Sudoku Euclidean distance trend at each step

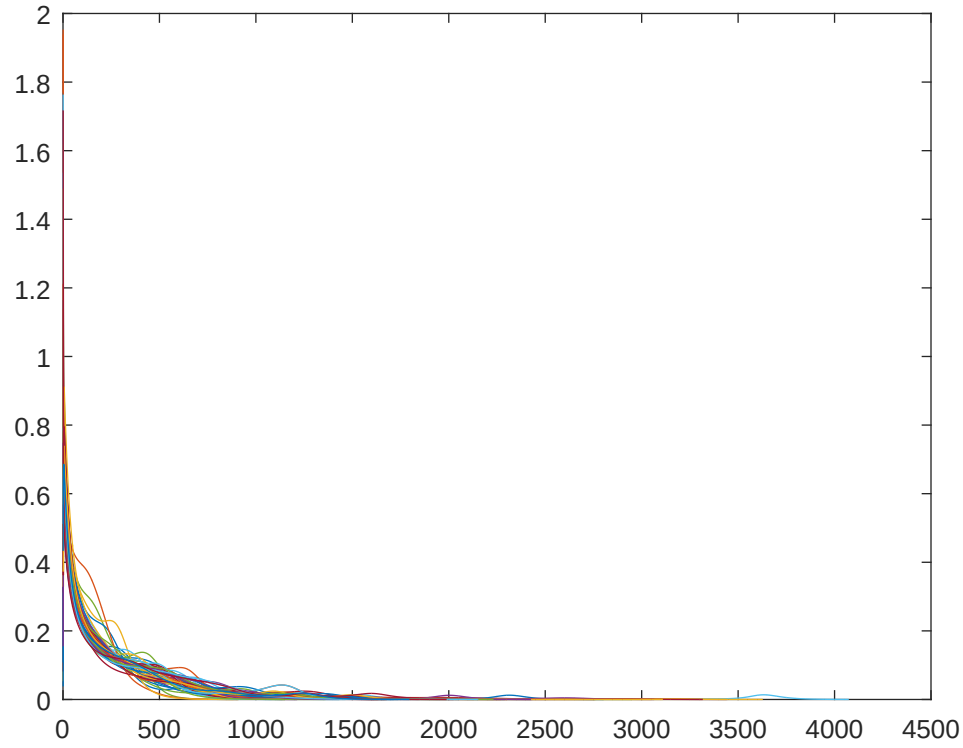

Figure 2: 50 Sudoku Average Local Consistency update variation each step

Evaluation

Completeness The algorithm does not always converge, in fact if we provide an empty Sudoku, the algorithm will stuck forever. After some tests, I noticed that it seems to get stuck when the update phase does not have enough motivation from the context to change the value of a wrong assignment i.e. the context say that this is a good assignment for the object, but in reality it is not. In fact, after running the algorithm on some non-convergent Sudoku, it continuously try to update the wrong cells but each time with a smaller and smaller improvement, this is because all the other correct assignments get stronger and stronger at each update. In fact this situation happens only with few inconsistent assignments, that's why the algorithm does not want to update the probabilities. In graph terms, seems to me that the algorithm has some difficulties to correct itself when the failing node of the search tree is in a very deep branch (analogy with the "local maximum" in optimization problems). A solution could be to randomize some values after the algorithm get stuck for a long period, to help the algorithm restart from another node of the search graph.

Optimality Surely it is not optimal, since it is not complete; when it provides the solution I sincerely don't know if it is optimal in terms of the nearest solution from the initial state in the search graph. Intuitively seems to me that it's going to guess the best label at the moment, this could be a reason of why it does not always converge and why it does not always provide the optimal solution.

Temporal Complexity Temporal complexity is somehow a problem here, because if we consider the labels structure for each cell we have $n = 81 \cdot 9 = 729$ probability cells to update at each iteration and one single update iteration requires a lot of computation (we have to find the max probable label for each cell etc.). In

fact, the algorithm is very slow compared to the backtracking solution, that's why I preferred to rewrite this solution in C instead of C#. The temporal complexity of this algorithm is intrinsic in the method invented by Rosenfeld and [...], there might be some smart tricks to speed up the time and improve the solution, but it would require some technical analysis.

Spatial Complexity The spatial complexity is not a big deal, we have to store 9×9 cells for the objects and for each cell we need an array of 9 double values to store the probability of the assignment for the given label.

Considerations

This method seems to heavily rely on contextual information, since at each step the algorithm update the probability of a particular label of a particular cell by looking at all the other cells. In fact if we supply to the algorithm an empty Sudoku, it will not solve it at all since the update phase will not detect any contextual information, a consequence of the latter consideration is that the relaxation labelling algorithm needs an initialization to proceed. After some runs on different Sudoku I noticed that the C# solution is very slow (a difficult Sudoku might require up to 20 minutes or more) that's why I ended up with rewriting the whole code into C and thanks to the -O3 flag during compiling it will require a lot less time, we can drop down the time from 20 minutes to 20 seconds. Actually if we use float numbers instead double numbers the time drops down even more, for example the assignment sudoku can be solved in 3 seconds or less.

Conclusion

I think that the two approaches are very different. The first one seems to me more deterministic and specialized, in the sense that we have much more information of the problem, we somehow are more expert on that particular problem so we developed more heuristics to find the best strategy for the search algorithm. In fact, we improved a simple backtracking solution (which for me it is a brute-force-like approach) by adding more heuristics like minimum remaining values and forward checking. I think that this comes from the fact that Sudoku is part of some class of problems that are deterministic and somehow very simple, that's why there are a lot of smart solutions and heuristics that have been developed, maybe because it could be reduced to a well know problem that has been studied very deeply.
The second approach seems to me more probabilistic-based, context-aware and general. This solution requires less knowledge of the problem, and provided that there is an initialization, the algorithm is somehow "finding good guesses" itself. I am actually not helping the algorithm by maintaining some special structures or hinting some good guesses coming from my specific knoledge of the problem, it goes by itself. It's like (pass me the term) an independent intelligent unit that his only purpose is solving this kind of problems, it's somehow aware of the task that has to do.

Student: Francesco Pelosin
Email: 839220@stud.unive.it *OR* pelosinfrancesco@live.it
Id: 839220