TDT4136 - Assignment 3
Constraint Satisfaction Problems

Arthur Testard - id: 105022

1 CSPs, Backtracking and AC-3

For this assignment, we want to implement a backtracking search in order to solve CSPs problem as it is described in Chapter 5.3 of [RN21]. We give the figure 5.5 of the book on Figure 1 which describes the logic of the backtracking search.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment then
          add {var = value} to assignment
          inferences ← INFERENCE(csp, var, value)
          if inferences ≠ failure then
             add inferences to assignment
             result ← BACKTRACK(assignment, csp)
             if result ≠ failure then
                return result
      remove {var = value} and inferences from assignment
  return failure
```

Figure 1: A simple backtracking algorithm for constraint satisfaction problem

We have also to specify what the following functions do: SELECT_UNASSIGNED_VARIABLE, ORDER_DOMAIN_VALUES and INFERENCES.

1.1 Select Unassigned Variable

We have to return, for every call of SELECT_UNASSIGNED_VARIABLE, which variable have been assigned or have not. An unassigned variable would have more than one value. Thus, SELECT_UNASSIGNED_VARIABLE select the first variable which has strictly more than one value associate too. Like this we are processing through the graph always by the same order.

1.2 Order Domain Values

We want to select the right order of unassigned values. To do so, the simplest strategy is to choose the value in the order of the domain values.

1.3 Inference

This function works the same way as the arc consistency algorithm AC-3 described in Figure 5.3 of [RN21]. We give the algorithm on Figure 2.

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REVISE(csp, Xi, Xj) then
      if size of Di = 0 then return false
      for each Xk in Xi.NEIGHBORS - {Xj} do
        add (Xk, Xi) to queue
  return true
--------------------------------------------------------------------------------
function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
  revised ← false
  for each x in Di do
    if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
      delete x from Di
      revised ← true
  return revised
```

Figure 2: The arc-consistency algorithm AC-3

In our implementation, when we want to get the neighbours, we get them depending on the different constraints. If we want the neighbours of X, we get all the arcs that could be visited. Those arcs depends on the constraints between every values of X. So, for any x in D_i, if we want that no value y in D_j allows (x, y) to satisfy the constraints between X_i and X_j, that means that we want, for y value to not be in the current assignment. If it is, we delete it from the current assignment, and then add to the queue, all of neighbors of the current assignment, even those which have already been checked. Our current value failed maybe because last one.

1.4 Implementation

The implementation of this program gives a solution for the map colouring CSP from the textbook (Chapter 5.1.1, p. 165). This solution is given on Figure 3:

```
>>> import Assignment as a
>>> map = a.create_map_coloring_csp()
>>> print(map.backtracking_search())
{'WA': ['red'],
    'NT': ['green'],
    'Q': ['red'],
    'NSW': ['green'],
    'V': ['red'],
    'SA': ['blue'],
    'T': ['red']}
```

Figure 3: Result of the implementation of the backtracking search on the map coloring problem

2 Sudoku boards as CSPs

Our implementation of the CSP solver based on backtracking search and AC-3, gives the solutions presented on Figure 4 for the Sudoku given in files `easy.txt`, `medium.txt`, `hard.txt` and `veryhard.txt`.

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

(a) Solution of `easy.txt`

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

(b) Solution of `medium.txt`

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

(c) Solution of `hard.txt`

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

(d) Solution of `veryhard.txt`

Figure 4: Solutions of the different sudokus given by our backtracking implementation

Those results have been obtained with only 1 call of backtrack function and no failure of it for the board of `easy.txt`, 3 calls and no failure for the board of `medium.txt`, 12 calls and 4 failures for the board of `hard.txt` and 68 calls and 57 failures for the board of `very.hard.txt`. We quickly notice that these results are linked to the difficulty of the board. More the board is difficult, more the algorithm would fail or recall itself.

We can now explain, in a sense these results. Figure 5 shows for the four sudokus board, the number of values which are arc-consistent at the beginning. For example we can see why `easy.txt` board only need one call of itself to be solved. It shows that `easy.txt`'s board admit just one arc-consistent solution, make it easy to solve. On the other hand, we can see that `very.hard.txt` board allows more arc-consistent solutions (up to 6 on some cells).

To get all the possibilities existing on each board we have to multiply by itself all the values of each board. Thus, we get the plot on Figure 6 which shows the number of possibilities depending on the board. We get approximately, in the difficulty order, 1, 3439853568 ($\approx 10^9$), 148618787703226368 ($\approx 10^{17}$) and 8036474935754883072 ($\approx 10^{18}$) arc-consistent boards possibilities.

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

(a) Possibilities on each cells of `easy.txt`

3	1	4	2	1	1	2	2	4
3	2	3	2	1	1	1	2	1
2	1	2	1	1	1	3	1	3
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
2	3	2	1	1	1	2	1	3
1	1	1	1	1	1	1	1	1
2	2	3	1	1	1	2	1	2

(b) Possibilities on each cells of `medium.txt`

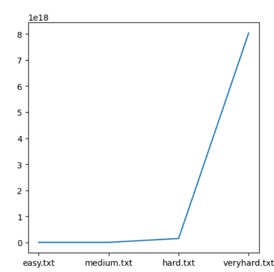
1	3	2	2	1	3	1	4	1
4	4	3	4	1	6	4	4	2
1	4	1	2	3	1	1	3	3
2	5	1	1	4	1	2	4	2
1	1	2	2	2	3	3	1	1
2	4	3	1	3	1	1	4	2
2	3	1	1	3	3	1	4	1
4	4	3	5	1	4	2	4	4
1	3	1	4	1	3	2	3	1

(c) Possibilities on each cells of `hard.txt`

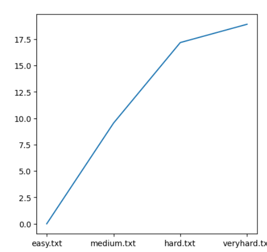
4	1	3	1	4	1	2	1	3
5	3	3	1	4	1	4	5	6
1	4	4	3	3	3	4	5	1
5	1	4	2	3	3	4	1	4
4	3	1	1	3	1	1	4	3
1	3	3	1	3	1	3	3	1
3	1	5	4	6	3	5	1	3
3	5	1	4	1	3	1	3	2
4	5	1	3	1	2	1	3	2

(d) Possibilities on each cells of `veryhard.txt`

Figure 5: Number of possibilities that are arc-consistent on each different cells of the different sudokus at the beginning



(a) Normal scale



(b) Logarithm scale

Figure 6: Arc-consistent possibilities depending on each board

3 To try to go further

One intuitive factor about the difficulty, and thus the complexity of our algorithm resolving Sudokus, would be the number of empty cells at the beginning. If we look on how much there is for each, we have: 49 for `easy.txt`'s board, 48 for `medium.txt`'s board, 53 for `hard.txt`'s board and 55 for `veryhard.txt`'s board. Because `easy.txt`'s board has more empty cells than `medium.txt`'s one. Regarding this results, we can conclude that there is a impact of the number of cells and the difficulty that the program has to resolve the different boards, but it is not sufficient.

We can say something similar about the constraints made by our CSP problem. As we can see on Figure 7, the difficulty (almost) increases with number of constraints which have a size equal to 72. This information seems a bit more relevant than just the number of zeros, because the constraints take into consideration the empty cells, and, the distribution of the values on the board. We look now on the number of constraints of a sudoku board.

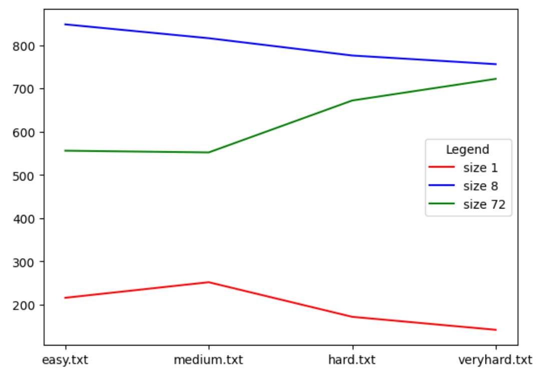


Figure 7: Size of initial constraints depending on the difficulty of the board

If we chose a random cell, this cell constraint depends on the 8 other cells in its row, 8 more in its column and 4 more in the 3 by 3 square of the cell (we can show that, for any cell, if we take out the cell and the cells associate to its column and row, there is left 4 cells in its grid). Thus a cell constraint depends on $8 + 8 + 4 = 20$ other cells. Because we have $9 * 9 = 81$ cells, we then have $20 * 81 = 1620$ constraints in a Sudoku board. These constraints are divided in 3: those which just has 1, 8 and $8 * 9 = 72$.

We can then show a funny result, which probably does not involve the difficulty of a Sudoku board. If we look on the number of 72-constraints cells which are in the 4 ones left in the grid, after deleting the concerned cell and those which are in the same column and row, and we plot it depending on the difficulty board, we have then, Figure 7. We can see that that number increases with the difficulty. It sounds fair but it could be a coincidence.

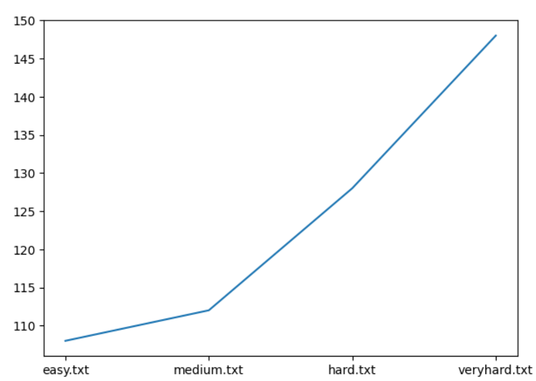


Figure 8: Number, for every cell, of 72-constraints which are in the same grid than the concerned cell, but not in its column or row

References

[RN21] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach, 4th us ed. *University of California, Berkeley*, 2021.