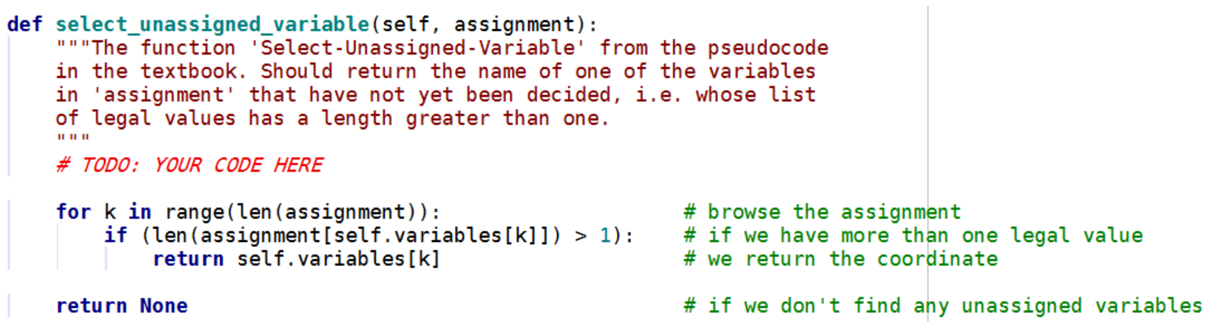
<u>Assignment 3</u>

Louka CHAPUS

1. <u>CSPs, Backtracking and AC-3 functions</u>

1.1 <u>Select-Unassigned-Variable fuction:</u>

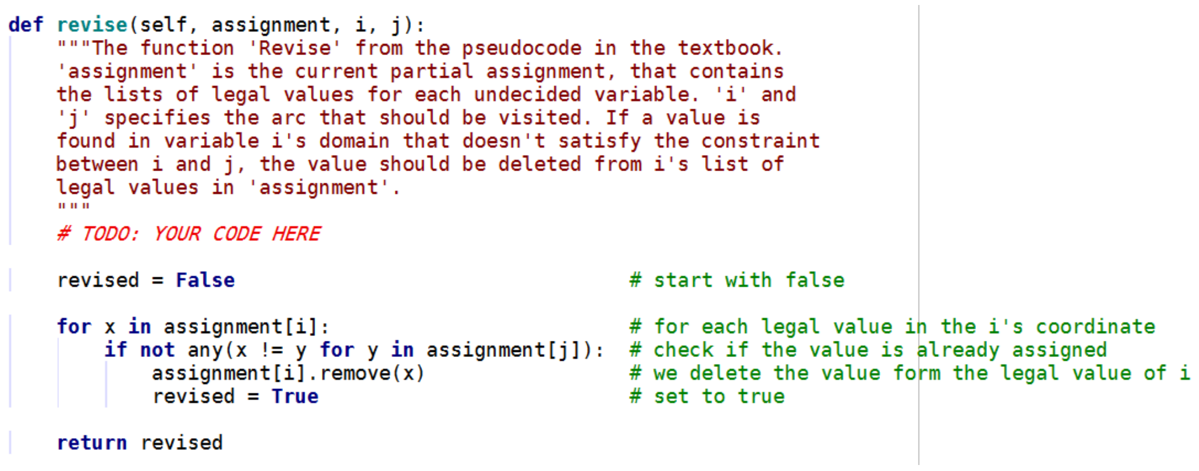
We need to browse the entire assignment and for each coordinate we look how many legal variables are possible. If there is more than one, this means that the variable is unassigned. And if we browse the entire assignment without finding any unassigned variable then we return none, it means that we have assigned all the variable and the search is finish.



```python
def select_unassigned_variable(self, assignment):
    """The function 'Select-Unassigned-Variable' from the pseudocode
    in the textbook. Should return the name of one of the variables
    in 'assignment' that have not yet been decided, i.e. whose list
    of legal values has a length greater than one.
    """
    # TODO: YOUR CODE HERE

    for k in range(len(assignment)):                     # browse the assignment
        if (len(assignment[self.variables[k]]) > 1):     # if we have more than one legal value
            return self.variables[k]                     # we return the coordinate

    return None                                          # if we don't find any unassigned variables
```

1.2 <u>Revise function:</u>

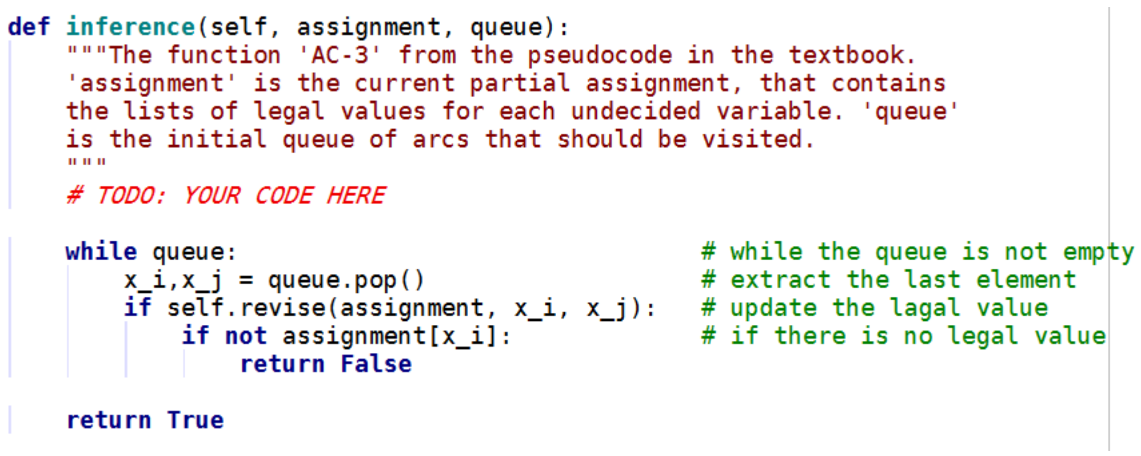
Then we need to have the revise function to see if the values are legal or not. If there are no legal value, we return false. But if we find value then we remove them from the other legal values and return true.



```python
def revise(self, assignment, i, j):
    """The function 'Revise' from the pseudocode in the textbook.
    'assignment' is the current partial assignment, that contains
    the lists of legal values for each undecided variable. 'i' and
    'j' specifies the arc that should be visited. If a value is
    found in variable i's domain that doesn't satisfy the constraint
    between i and j, the value should be deleted from i's list of
    legal values in 'assignment'.
    """
    # TODO: YOUR CODE HERE

    revised = False                                  # start with false

    for x in assignment[i]:                          # for each legal value in the i's coordinate
        if not any(x != y for y in assignment[j]):   # check if the value is already assigned
            assignment[i].remove(x)                  # we delete the value form the legal value of i
            revised = True                           # set to true

    return revised
```

1.3 <u>Inference function:</u>

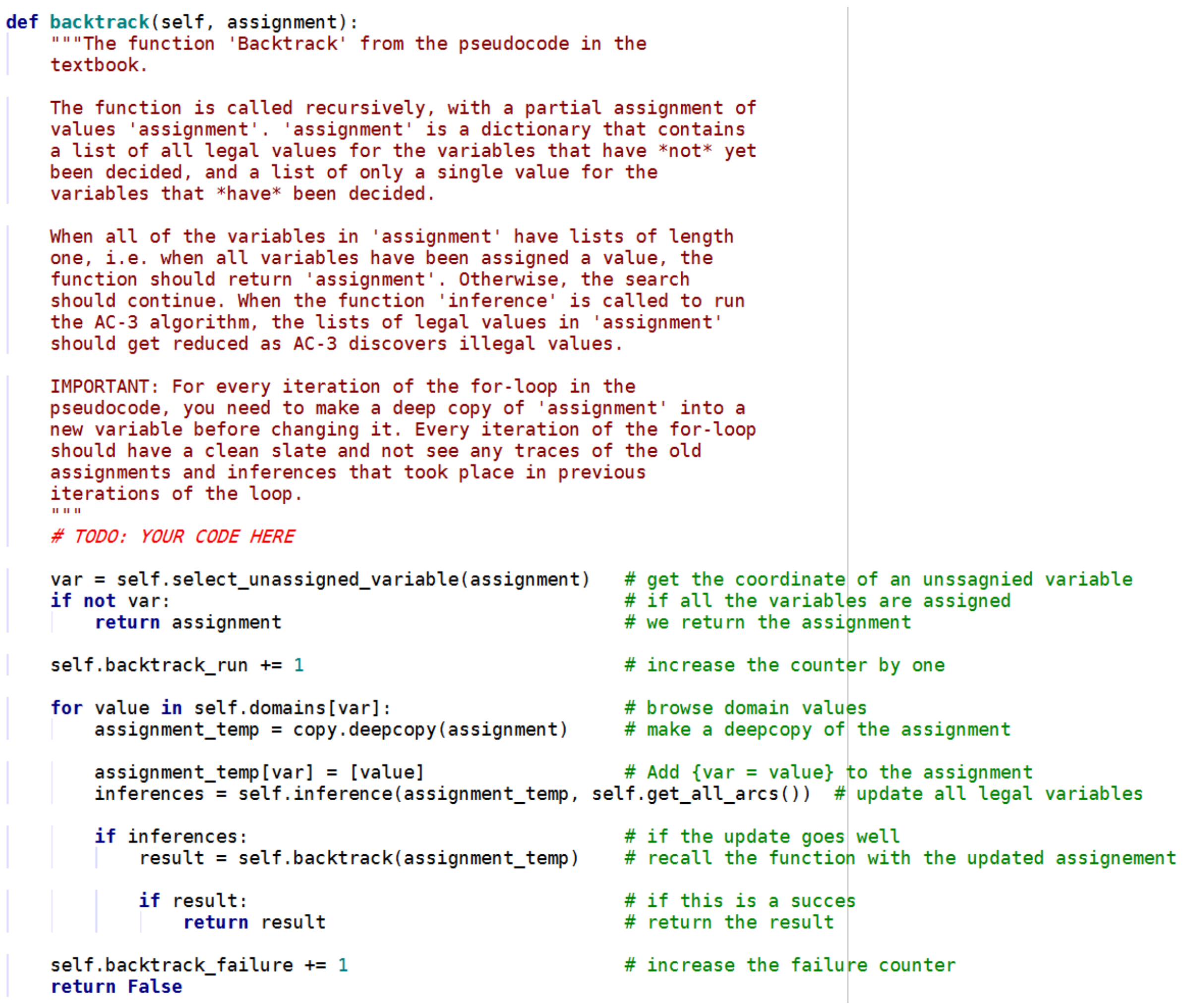
We generalize the previous function to handle all the neighbour (row, column and square) of a point and update the legal value of all neighbours. We just need to make a loop and test all the elements with the previous function.



```python
def inference(self, assignment, queue):
    """The function 'AC-3' from the pseudocode in the textbook.
    'assignment' is the current partial assignment, that contains
    the lists of legal values for each undecided variable. 'queue'
    is the initial queue of arcs that should be visited.
    """
    # TODO: YOUR CODE HERE

    while queue:                                    # while the queue is not empty
        x_i,x_j = queue.pop()                       # extract the last element
        if self.revise(assignment, x_i, x_j):       # update the lagal value
            if not assignment[x_i]:                 # if there is no legal value
                return False

    return True
```

1.4 <u>Backtrack function:</u>

It is the most important function because it uses all the previous function to complete the sudoku. I use the pseudocode in the textbook to make it. The algorithm essentialy try an unassigned value and then look if the rest of the board is doable, if this is the case we go on the next unassigned value and do the same thing until we reach the end or a dead end. If we have a dead end we go on the previous version of the board and we increase the preious unassigned value.

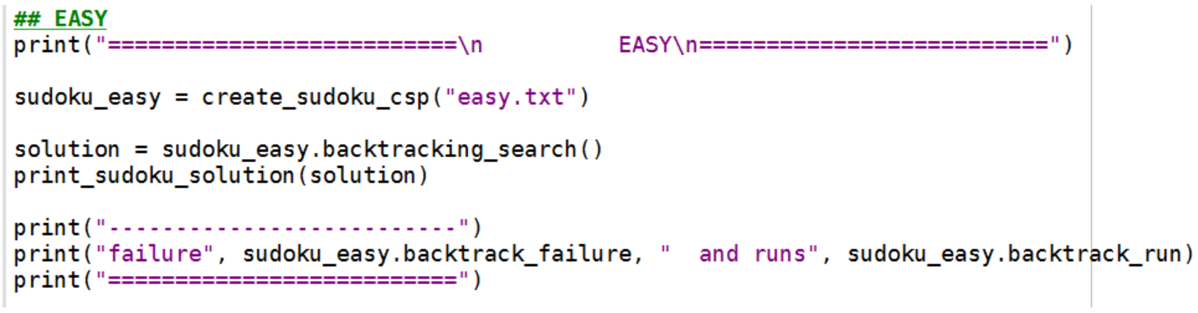


```python
def backtrack(self, assignment):
    """The function 'Backtrack' from the pseudocode in the
    textbook.

    The function is called recursively, with a partial assignment of
    values 'assignment'. 'assignment' is a dictionary that contains
    a list of all legal values for the variables that have *not* yet
    been decided, and a list of only a single value for the
    variables that *have* been decided.

    When all of the variables in 'assignment' have lists of length
    one, i.e. when all variables have been assigned a value, the
    function should return 'assignment'. Otherwise, the search
    should continue. When the function 'inference' is called to run
    the AC-3 algorithm, the lists of legal values in 'assignment'
    should get reduced as AC-3 discovers illegal values.

    IMPORTANT: For every iteration of the for-loop in the
    pseudocode, you need to make a deep copy of 'assignment' into a
    new variable before changing it. Every iteration of the for-loop
    should have a clean slate and not see any traces of the old
    assignments and inferences that took place in previous
    iterations of the loop.
    """
    # TODO: YOUR CODE HERE

    var = self.select_unassigned_variable(assignment)   # get the coordinate of an unssagnied variable
    if not var:                                         # if all the variables are assigned
        return assignment                               # we return the assignment

    self.backtrack_run += 1                             # increase the counter by one

    for value in self.domains[var]:                     # browse domain values
        assignment_temp = copy.deepcopy(assignment)     # make a deepcopy of the assignment

        assignment_temp[var] = [value]                  # Add {var = value} to the assignment
        inferences = self.inference(assignment_temp, self.get_all_arcs())  # update all legal variables

        if inferences:                                  # if the update goes well
            result = self.backtrack(assignment_temp)    # recall the function with the updated assignement

            if result:                                  # if this is a succes
                return result                           # return the result

    self.backtrack_failure += 1                         # increase the failure counter
    return False
```

2. Result with sudoku board

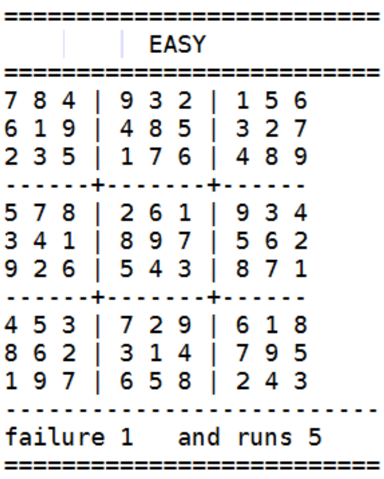
2.1 Easy board:

To solve the board, we just need to call the backtracking_search function after we have made a CSP object with the right data. And I just print the number times the function backtrack have failed and runs.



```
## EASY
print("==========================\n          EASY\n==========================")

sudoku_easy = create_sudoku_csp("easy.txt")

solution = sudoku_easy.backtracking_search()
print_sudoku_solution(solution)

print("--------------------------")
print("failure", sudoku_easy.backtrack_failure, "  and runs", sudoku_easy.backtrack_run)
print("==========================")
```

I got the following result.

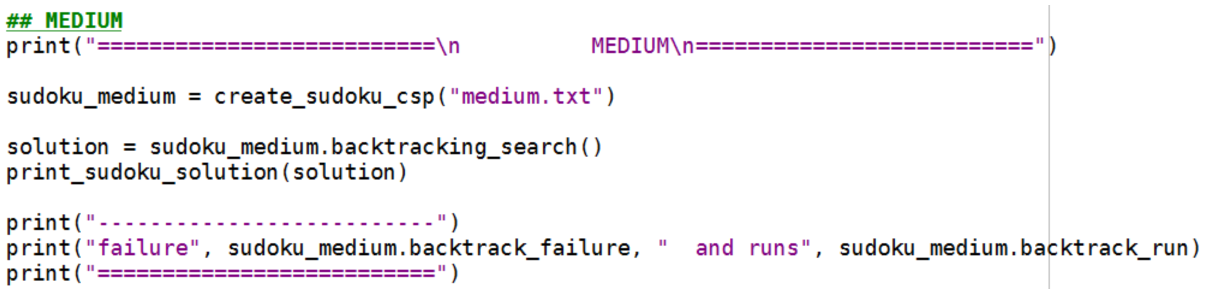


```
==========================
          EASY
==========================
7 8 4 | 9 3 2 | 1 5 6
6 1 9 | 4 8 5 | 3 2 7
2 3 5 | 1 7 6 | 4 8 9
------+-------+------
5 7 8 | 2 6 1 | 9 3 4
3 4 1 | 8 9 7 | 5 6 2
9 2 6 | 5 4 3 | 8 7 1
------+-------+------
4 5 3 | 7 2 9 | 6 1 8
8 6 2 | 3 1 4 | 7 9 5
1 9 7 | 6 5 8 | 2 4 3
--------------------------
failure 1   and runs 5
==========================
```

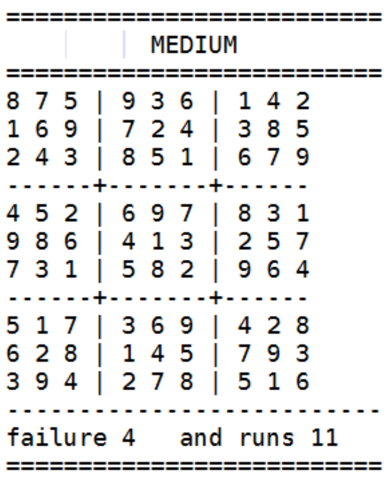
We got only one failure and 5 runs of the function. So, we see that the algorithm is very efficient.

2.2 Medium board:

It is the same process; we just need to change the name of the txt when we create the CSP object. And I got the following implementation and result.



```
## MEDIUM
print("==========================\n          MEDIUM\n==========================")

sudoku_medium = create_sudoku_csp("medium.txt")

solution = sudoku_medium.backtracking_search()
print_sudoku_solution(solution)

print("--------------------------")
print("failure", sudoku_medium.backtrack_failure, "  and runs", sudoku_medium.backtrack_run)
print("==========================")
```

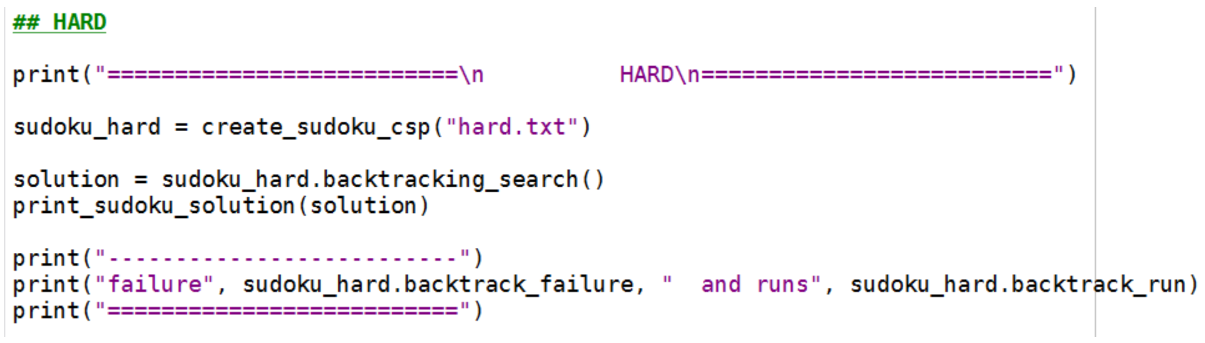


```
==========================
        MEDIUM
==========================
8 7 5 | 9 3 6 | 1 4 2
1 6 9 | 7 2 4 | 3 8 5
2 4 3 | 8 5 1 | 6 7 9
------+-------+------
4 5 2 | 6 9 7 | 8 3 1
9 8 6 | 4 1 3 | 2 5 7
7 3 1 | 5 8 2 | 9 6 4
------+-------+------
5 1 7 | 3 6 9 | 4 2 8
6 2 8 | 1 4 5 | 7 9 3
3 9 4 | 2 7 8 | 5 1 6
-------------------------
failure 4   and runs 11
==========================
```

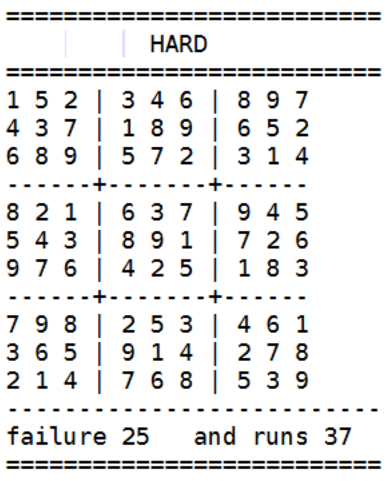
We also have a few failures and runs, it's because the sudoku is simple to solve at that point.

2.3 Hard board:

Again, we just must change the name. And we got the following result.



```
## HARD

print("==========================\n          HARD\n==========================")

sudoku_hard = create_sudoku_csp("hard.txt")

solution = sudoku_hard.backtracking_search()
print_sudoku_solution(solution)

print("-------------------------")
print("failure", sudoku_hard.backtrack_failure, "  and runs", sudoku_hard.backtrack_run)
print("==========================")
```

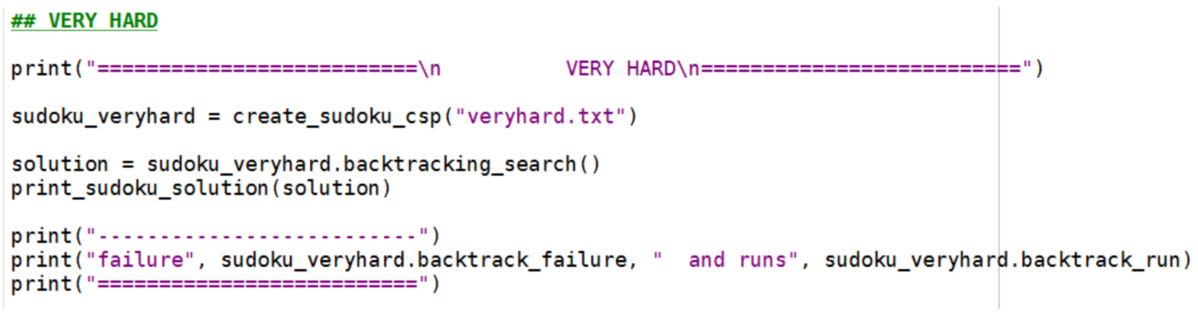


```
==========================
        HARD
==========================
1 5 2 | 3 4 6 | 8 9 7
4 3 7 | 1 8 9 | 6 5 2
6 8 9 | 5 7 2 | 3 1 4
------+-------+------
8 2 1 | 6 3 7 | 9 4 5
5 4 3 | 8 9 1 | 7 2 6
9 7 6 | 4 2 5 | 1 8 3
------+-------+------
7 9 8 | 2 5 3 | 4 6 1
3 6 5 | 9 1 4 | 2 7 8
2 1 4 | 7 6 8 | 5 3 9
-------------------------
failure 25   and runs 37
==========================
```

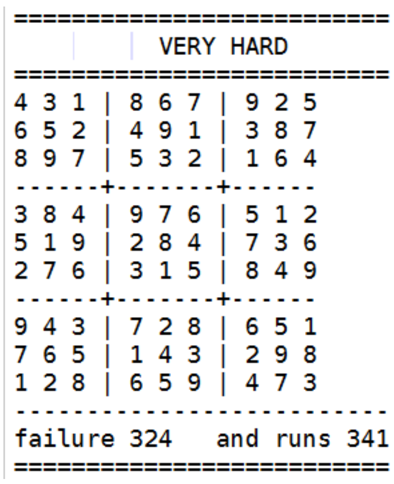
The board is harder this time so that's normal we got more failures and runs. But we got the correct answer from the algorithm, and we have the result almost instantly.

2.4 Hard board:

Same thing for the last one, we just change the name and runs the algorithm. We have the following code and result.



```
## VERY HARD

print("==========================\n          VERY HARD\n==========================")

sudoku_veryhard = create_sudoku_csp("veryhard.txt")

solution = sudoku_veryhard.backtracking_search()
print_sudoku_solution(solution)

print("--------------------------")
print("failure", sudoku_veryhard.backtrack_failure, "  and runs", sudoku_veryhard.backtrack_run)
print("==========================")
```



```
==========================
          VERY HARD
==========================
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
--------------------------
failure 324   and runs 341
==========================
```

Here we got a lot more failures and runs because we have few numbers in the board at the start. So, there are more possibilities and there are more try. But it is still very fast, only few seconds, compared if we have tried each board possible with a naive method.