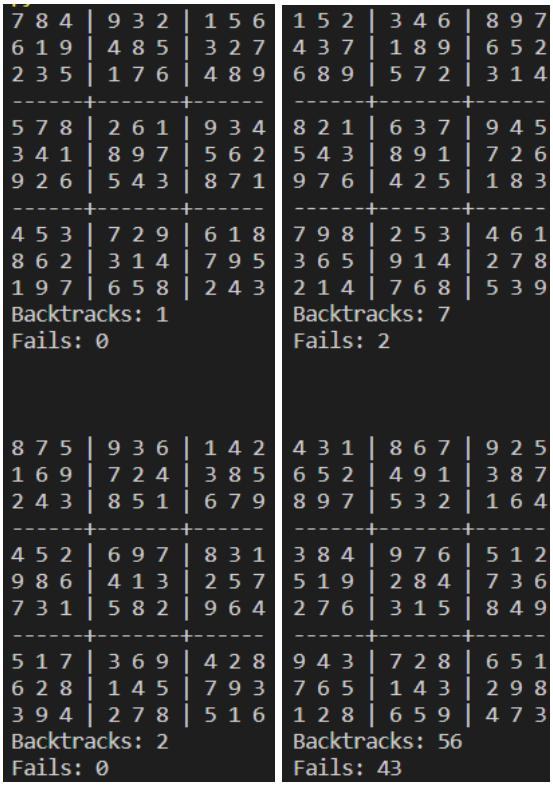
Assignment 4



Easy/hard

Medium/veryhard

The amount of backtrack and fails for higher difficulty increases as expected. Using a min select unassigned variable function decreased it further. The solutions for the first two boards are very low, and I expected it to have more backtracks.

```
CSP Assignment
# Original code by Håkon Måløy
# Updated by Xavier Sánchez Díaz

import copy
from itertools import product as prod


class CSP:
    def __init__(self):
        # self.variables is a list of the variable names in the CSP
        self.variables = []

        # self.domains is a dictionary of domains (lists)
        self.domains = {}

        # self.constraints[i][j] is a list of legal value pairs for
        # the variable pair (i, j)
        self.constraints = {}

        self.backtracks = 0
        self.fails = 0

    def add_variable(self, name: str, domain: list):
        """Add a new variable to the CSP.

        Parameters
        ----------
        name : str
            The name of the variable to add
        domain : list
            A list of the legal values for the variable
        """
        self.variables.append(name)
        self.domains[name] = list(domain)
        self.constraints[name] = {}

    def get_all_possible_pairs(self, a: list, b: list) -> list[tuple]:
        """Get a list of all possible pairs (as tuples) of the values in
        lists 'a' and 'b', where the first component comes from list
        'a' and the second component comes from list 'b'.

        Parameters
        ----------
        a : list
            First list of values
        b : list
            Second list of values
```

```
        Returns
        -------
        list[tuple]
            List of tuples in the form (a, b)
        """
        return prod(a, b)

    def get_all_arcs(self) -> list[tuple]:
        """Get a list of all arcs/constraints that have been defined in
        the CSP.

        Returns
        -------
        list[tuple]
            A list of tuples in the form (i, j), which represent a
            constraint between variable `i` and `j`
        """
        return [(i, j) for i in self.constraints for j in self.constraints[i]]

    def get_all_neighboring_arcs(self, var: str) -> list[tuple]:
        """Get a list of all arcs/constraints going to/from variable 'var'.

        Parameters
        ----------
        var : str
            Name of the variable

        Returns
        -------
        list[tuple]
            A list of all arcs/constraints in which `var` is involved
        """
        return [(i, var) for i in self.constraints[var]]

    def add_constraint_one_way(self, i: str, j: str,
                               filter_function: callable):
        """Add a new constraint between variables 'i' and 'j'. Legal
        values are specified by supplying a function 'filter_function',
        that should return True for legal value pairs, and False for
        illegal value pairs.

        NB! This method only adds the constraint one way, from i -> j.
        You must ensure to call the function the other way around, in
        order to add the constraint the from j -> i, as all constraints
        are supposed to be two-way connections!

        Parameters
        ----------
        i : str
```

```
            Name of the first variable
        j : str
            Name of the second variable
        filter_function : callable
            A callable (function name) that needs to return a boolean.
            This will filter value pairs which pass the condition and
            keep away those that don't pass your filter.
        """
        if j not in self.constraints[i]:
            # First, get a list of all possible pairs of values
            # between variables i and j
            self.constraints[i][j] = self.get_all_possible_pairs(
                                        self.domains[i],
                                        self.domains[j])

        # Next, filter this list of value pairs through the function
        # 'filter_function', so that only the legal value pairs remain
        self.constraints[i][j] = list(filter(lambda
                                             value_pair:
                                             filter_function(*value_pair),
                                             self.constraints[i][j]))

    def add_all_different_constraint(self, var_list: list):
        """Add an Alldiff constraint between all of the variables in the
        list provided.

        Parameters
        ----------
        var_list : list
            A list of variable names
        """
        for (i, j) in self.get_all_possible_pairs(var_list, var_list):
            if i != j:
                self.add_constraint_one_way(i, j, lambda x, y: x != y)

    def backtracking_search(self):
        """This functions starts the CSP solver and returns the found
        solution.
        """
        # Make a so-called "deep copy" of the dictionary containing the
        # domains of the CSP variables. The deep copy is required to
        # ensure that any changes made to 'assignment' does not have any
        # side effects elsewhere.
        assignment = copy.deepcopy(self.domains)

        # Run AC-3 on all constraints in the CSP, to weed out all of the
        # values that are not arc-consistent to begin with
        self.inference(assignment, self.get_all_arcs())
```

```
        # Call backtrack with the partial assignment 'assignment'
        return self.backtrack(assignment)

    def finish(self, assignment):
        """Checks if all the lists in assignment are of length one.

        Args:
            assignment (List): list of lists

        Returns:
            boolean: true if all lists in assignment are of length 1
        """
        for i in assignment.values():
            if len(i) > 1:
                return False
        return True

    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.
        """
        self.backtracks+=1
        if self.finish(assignment):
            # print(self.backtracks)
            return assignment


        edit = self.select_unassigned_variable(assignment)
```

```
        for value in assignment[edit]:

            ass_copy = copy.deepcopy(assignment)
            ass_copy[edit] = [value]
            inferences = self.inference(ass_copy, self.get_all_arcs())
            if inferences:
                result = self.backtrack(ass_copy)

                if result:
                    return result
                # ass_copy.remove(inferences)
            # assignment.remove()

        self.fails += 1
        return False

    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.
        """
        """Returns the shortest list in assignment.

        Returns:
            _type_: _description_
        """

        # for i in assignment:
        #     if len(assignment[i]) > 1:
        #         return i
        minval = 10000000
        for i in assignment:
            if minval > len(assignment[i]) > 1:
                min = i
                minval = len(assignment[i])
        return min
        # for i in self.variables:
        #     if len(assignment[i]) > 1:
        #         return i

        #     if len(i) == 1:
        #         assignment.remove(i)
        # return min(assignment, key=len)

    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.
        """
        while len(queue)>0:
            i, j = queue.pop()
            if self.revise(assignment,i,j):
                if len(assignment[i]) == 0:
                    return False
                for k in self.get_all_neighboring_arcs(i):
                    if k[0] != j:
                        queue.append([k[0],i])
        return True


    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'.
        """
        removed = False
        for x in assignment[i]:
            isLegal = False
            for y in assignment[j]:
                if (x,y) in self.constraints[i][j]:
                    isLegal = True
            if not isLegal:
                assignment[i].remove(x)
                removed = True
        return removed



def create_map_coloring_csp():
    """Instantiate a CSP representing the map coloring problem from the
    textbook. This can be useful for testing your CSP solver as you
    develop your code.
    """
    csp = CSP()
    states = ['WA', 'NT', 'Q', 'NSW', 'V', 'SA', 'T']
    edges = {'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],
             'NT': ['WA', 'Q'], 'NSW': ['Q', 'V']}
    colors = ['red', 'green', 'blue']
    for state in states:
        csp.add_variable(state, colors)
    for state, other_states in edges.items():
        for other_state in other_states:
```

```
            csp.add_constraint_one_way(state, other_state, lambda i, j: i !=
j)
            csp.add_constraint_one_way(other_state, state, lambda i, j: i !=
j)
    return csp


def create_sudoku_csp(filename: str) -> CSP:
    """Instantiate a CSP representing the Sudoku board found in the text
    file named 'filename' in the current directory.

    Parameters
    ----------
    filename : str
        Filename of the Sudoku board to solve

    Returns
    -------
    CSP
        A CSP instance
    """
    csp = CSP()
    board = list(map(lambda x: x.strip(), open(filename, 'r')))

    for row in range(9):
        for col in range(9):
            if board[row][col] == '0':
                csp.add_variable('%d-%d' % (row, col), list(map(str,
                                                                range(1,
10))))
            else:
                csp.add_variable('%d-%d' % (row, col), [board[row][col]])

    for row in range(9):
        csp.add_all_different_constraint(['%d-%d' % (row, col)
                                          for col in range(9)])
    for col in range(9):
        csp.add_all_different_constraint(['%d-%d' % (row, col)
                                          for row in range(9)])
    for box_row in range(3):
        for box_col in range(3):
            cells = []
            for row in range(box_row * 3, (box_row + 1) * 3):
                for col in range(box_col * 3, (box_col + 1) * 3):
                    cells.append('%d-%d' % (row, col))
            csp.add_all_different_constraint(cells)

    return csp
```

```
def print_sudoku_solution(solution):
    """Convert the representation of a Sudoku solution as returned from
    the method CSP.backtracking_search(), into a human readable
    representation.
    """
    for row in range(9):
        for col in range(9):
            print(solution['%d-%d' % (row, col)][0], end=" "),
            if col == 2 or col == 5:
                print('|', end=" "),
        print("")
        if row == 2 or row == 5:
            print('------+-------+------')



def print_all(filename):
    csp = create_sudoku_csp(filename)
    print_sudoku_solution(csp.backtracking_search())
    print('Backtracks: ' + str(csp.backtracks))
    print('Fails: ' + str(csp.fails))
    print('\n\n')

print_all('easy.txt')
print_all('medium.txt')
print_all('hard.txt')
print_all('veryhard.txt')




# p = create_map_coloring_csp()
# print(p)
# print(p.constraints)
# print(p.domains)
# print(p.variables)
# p2 = create_sudoku_csp('medium.txt')
# solution = p2.backtracking_search()
# print_sudoku_solution(solution)
# print(p2.constraints)
# print(p2.backtracks)
# print(p2.fails)

# p3 = create_sudoku_csp('easy.txt')
# solution3 = p3.backtracking_search()
# print_sudoku_solution(solution3)
# print(p3.backtracks)
# print(p3.fails)
```

Assignment 4