Michelle Lin (netid: ml7188) and Tianzuo Liu (netid: tl3119)

To run the program, in terminal (at the same path with source code file):

$python main.py (testfilename.txt)

Output for Input1.txt

9567
1085
10652

Output for Input2.txt

7483
7455
14938

```
# Author: Michelle Lin (netid: ml7188) and Tianzuo Liu (netid:
tl3119)
import argparse
from typing import List, Tuple


def minimum_remaining_values(variables, assigned, domains):
    unassigned_variables = [var for var in variables if var not
in assigned]
    if not unassigned_variables:
        return None

    min_length = len(domains[unassigned_variables[0]])
    min_variables = [unassigned_variables[0]]

    for var in unassigned_variables[1:]:
        if len(domains[var]) < min_length: # get the variable
with the minimum number of remaining values
            min_length = len(domains[var])
            min_variables = [var]
        # if there are multiple variables have same values
        elif len(domains[var]) == min_length:
            min_variables.append(var)

    return min_variables
```

```
# if multiple values are left after apply MRV, then apply
degree_heuristic
def degree_heuristics(variables, constraints, assigned,
min_variables):
    unassigned_variables = [var for var in variables if var not
in assigned]
    if not unassigned_variables:
        return None

    # a dictionary to count number of constraints in the default
constraints
    degrees = {var: 0 for var in unassigned_variables}

    for constraint in constraints:
        var1, var2 = constraint
        if var1 in min_variables and var2 in
unassigned_variables:
            degrees[var1] += 1
        if var1 in unassigned_variables and var2 in
min_variables:
            degrees[var2] += 1

    degree_variable = max(degrees, key=degrees.get)

    return degree_variable


def set_up_constraints(variables, unique_variables):
    # set constraints as List[Tuple[str, str]], ex: [(x1, x2),
(x1,x3)]
    constraints = []

    # set up Alldiff constraint (C1)
    for i in unique_variables:
        for j in unique_variables:
            if j != i:
                if ((i, j) not in constraints) and ((j, i) not
in constraints):
                    if i < j: # according to alphabetical order
so that no repeated constraints
                        constraints += [(i, j)]
                    else:
                        constraints += [(j, i)]

    # set up rest of the constraints
    rest_constraints = []
```

```
    # C2 (x4+x8 = 10*alpha+x13)
    rest_constraints += [tuple(sorted((variables[4-1],
variables[8-1])))]
    rest_constraints += [tuple(sorted((variables[4-1],
variables[13-1])))]
    rest_constraints += [tuple(sorted((variables[8-1],
variables[13-1])))]
    rest_constraints += [tuple(sorted(("alpha",
variables[13-1])))]
    rest_constraints += [tuple(sorted(("alpha",
variables[4-1])))]
    rest_constraints += [tuple(sorted(("alpha",
variables[8-1])))]

    # C3 (alpha+x3+x7 = 10*beta+x12)
    rest_constraints += [tuple(sorted(("alpha",
variables[3-1])))]
    rest_constraints += [tuple(sorted(("alpha",
variables[7-1])))]
    rest_constraints += [tuple(sorted(("alpha",
variables[12-1])))]
    rest_constraints += [tuple(sorted(("alpha", "beta")))]
    rest_constraints += [tuple(sorted((variables[3-1],
variables[7-1])))]
    rest_constraints += [tuple(sorted((variables[3-1],
variables[12-1])))]
    rest_constraints += [tuple(sorted((variables[3-1],
"beta")))]
    rest_constraints += [tuple(sorted((variables[7-1],
"beta")))]
    rest_constraints += [tuple(sorted((variables[7-1],
variables[12-1])))]
    rest_constraints += [tuple(sorted((variables[12-1],
"beta")))]

    # C4 (beta+x2+x6 = 10*gamma+x11)
    rest_constraints += [tuple(sorted(("beta",
variables[2-1])))]
    rest_constraints += [tuple(sorted(("beta",
variables[6-1])))]
    rest_constraints += [tuple(sorted(("beta",
variables[11-1])))]
    rest_constraints += [tuple(sorted(("beta", "gamma")))]
    rest_constraints += [tuple(sorted((variables[2-1],
variables[6-1])))]
```

```
    rest_constraints += [tuple(sorted((variables[2-1],
"gamma")))]
    rest_constraints += [tuple(sorted((variables[2-1],
variables[11-1])))]
    rest_constraints += [tuple(sorted((variables[6-1],
"gamma")))]
    rest_constraints += [tuple(sorted((variables[6-1],
variables[11-1])))]
    rest_constraints += [tuple(sorted((variables[11-1],
"gamma")))]

    # C5 (gamma+x1+x5 = x9*10+x10)
    rest_constraints += [tuple(sorted((variables[1-1],
"gamma")))]
    rest_constraints += [tuple(sorted((variables[5-1],
"gamma")))]
    rest_constraints += [tuple(sorted((variables[9-1],
"gamma")))]
    rest_constraints += [tuple(sorted((variables[10-1],
"gamma")))]
    rest_constraints += [tuple(sorted((variables[1-1],
variables[5-1])))]
    rest_constraints += [tuple(sorted((variables[1-1],
variables[10-1])))]
    rest_constraints += [tuple(sorted((variables[5-1],
variables[10-1])))]
    rest_constraints += [tuple(sorted((variables[1-1],
variables[9-1])))]
    rest_constraints += [tuple(sorted((variables[5-1],
variables[9-1])))]
    rest_constraints += [tuple(sorted((variables[10-1],
variables[9-1])))]

    for i in rest_constraints:
        if i not in constraints:
            constraints += [i]

    return constraints

def select_unassigned_variable(unique_variables, assignment,
domains, constraints):
    min_variables = minimum_remaining_values(unique_variables,
assignment, domains) # use MRV first
    if min_variables is None:
        return None
    else:
```

```
        if len(min_variables) > 1: # if more than one variable
selected by MRV, use degree heuristics
            degree_var = degree_heuristics(unique_variables,
constraints, assignment, min_variables)
            if degree_var is None:
                return None
            else:
                return degree_var
        else:
            return min_variables

def backtracking_search(variables, unique_variables, domains,
constraint, assignment, clean_variables):
    if len(assignment) == len(unique_variables): # if assignment
complete
        if check_completion(assignment, clean_variables):
            print("solution found!!!")
            print("assignment: ", assignment)
            generate_output_file(assignment, variables)
            return assignment
    current_variable =
select_unassigned_variable(unique_variables, assignment,
domains, constraint)
    if current_variable is not None:
        current_variable = current_variable[0] # current
variable return a list of len = 1
        for value in domains[current_variable]:
            if value not in assignment.values():  # Check if the
value is not already assigned
                new_assignment = assignment.copy()
                if consistent(current_variable, value,
new_assignment, variables, domains):
                    new_assignment[current_variable] = value
                    result = backtracking_search(variables,
unique_variables, domains, constraint, new_assignment,
clean_variables)
                    if result is not None:
                        return result
                assignment[current_variable] = None
    return None

def consistent(current_variable, value, assignment, variables,
domains):
    assignment[current_variable] = value

    # C1: Alldiff(letters)
```

```
    if len(set(assignment.values())) != len(assignment):
        return False

    if current_variable == variables[4-1] or current_variable ==
variables[8-1] or current_variable == variables[13-1]:
        # C2: (x4+x8 = 10*alpha+x13)
        if variables[4-1] in assignment: # if variable already
assigned, domain is the assigned value
            x4_domain = [assignment[variables[4-1]]]
        else: # if the variable not yet assigned, domain is the
domain values
            x4_domain = domains[variables[4-1]]
        if variables[8-1] in assignment:
            x8_domain = [assignment[variables[8-1]]]
        else:
            x8_domain = domains[variables[8-1]]
        if variables[13-1] in assignment:
            x13_domain = [assignment[variables[13-1]]]
        else:
            x13_domain = domains[variables[13-1]]

        alpha_domain = domains["alpha"]

        for four in x4_domain: # check if there is a combination
on the domain that satisfies constraints
            for eight in x8_domain:
                for thirteen in x13_domain:
                    for a in alpha_domain:
                        if four + eight == 10 * a + thirteen:
                            return True
        return False

    if current_variable == variables[3-1] or current_variable ==
variables[7-1] or current_variable == variables[12-1]:
        # C3: (alpha+x3+x7 = 10*beta+x12)
        if variables[3-1] in assignment:
            x3_domain = [assignment[variables[3-1]]]
        else:
            x3_domain = domains[variables[3-1]]
        if variables[7-1] in assignment:
            x7_domain = [assignment[variables[7-1]]]
        else:
            x7_domain = domains[variables[7-1]]
        if variables[12-1] in assignment:
            x12_domain = [assignment[variables[12-1]]]
        else:
```

```
            x12_domain = domains[variables[12-1]]

        alpha_domain = domains["alpha"]
        beta_domain = domains["beta"]

        for a in alpha_domain:
            for three in x3_domain:
                for seven in x7_domain:
                    for b in beta_domain:
                        for twelve in x12_domain:
                            if a + three + seven == 10 * b +
twelve:
                                return True
        return False

    if current_variable == variables[2-1] or current_variable ==
variables[6-1] or current_variable == variables[11-1]:
        # C4: (beta+x2+x6 = 10*gamma+x11)
        if variables[2-1] in assignment:
            x2_domain = [assignment[variables[2-1]]]
        else:
            x2_domain = domains[variables[2-1]]
        if variables[6-1] in assignment:
            x6_domain = [assignment[variables[6-1]]]
        else:
            x6_domain = domains[variables[6-1]]
        if variables[11-1] in assignment:
            x11_domain = [assignment[variables[11-1]]]
        else:
            x11_domain = domains[variables[11-1]]

        beta_domain = domains["beta"]
        gamma_domain = domains["gamma"]

        for b in beta_domain:
            for two in x2_domain:
                for six in x6_domain:
                    for c in gamma_domain:
                        for eleven in x11_domain:
                            if b + two + six == 10 * c + eleven:
                                return True
        return False

    if current_variable == variables[1-1] or current_variable ==
variables[5-1] or current_variable == variables[10-1] or
current_variable == variables[9-1]:
```

```
        # C5: (gamma+x1+x5 = x9*10+x10)
        if variables[1-1] in assignment:
            x1_domain = [assignment[variables[1-1]]]
        else:
            x1_domain = domains[variables[1-1]]
        if variables[5-1] in assignment:
            x5_domain = [assignment[variables[5-1]]]
        else:
            x5_domain = domains[variables[5-1]]
        if variables[10-1] in assignment:
            x10_domain = [assignment[variables[10-1]]]
        else:
            x10_domain = domains[variables[10-1]]
        if variables[9-1] in assignment:
            x9_domain = [assignment[variables[9-1]]]
        else:
            x9_domain = domains[variables[9-1]]

        gamma_domain = domains["gamma"]

        for c in gamma_domain:
            for one in x1_domain:
                for five in x5_domain:
                    for ten in x10_domain:
                        for nine in x9_domain:
                            if c + one + five == nine * 10 +
ten:
                                return True
        return False

def check_completion(assignment, clean_variables): # function
checked
    line1_value = (assignment[clean_variables[1-1]]*1000
                   +assignment[clean_variables[2-1]]*100
                   +assignment[clean_variables[3-1]]*10
                   +assignment[clean_variables[4-1]])
    line2_value = (assignment[clean_variables[5-1]]*1000
                   +assignment[clean_variables[6-1]]*100
                   +assignment[clean_variables[7-1]]*10
                   +assignment[clean_variables[8-1]])
    line3_value = (assignment[clean_variables[9-1]]*10000
                   +assignment[clean_variables[10-1]]*1000
                   +assignment[clean_variables[11-1]]*100
                   +assignment[clean_variables[12-1]]*10
                   +assignment[clean_variables[13-1]])
    if line1_value + line2_value == line3_value:
```

```
            return True
        else:
            return False

def generate_output_file(assignment, variables):
    # Parse command-line arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('filename')
    cmdline = parser.parse_args()

    # Read input file and create initial and goal states
    with open(cmdline.filename, 'r') as file:
        lines = file.read().splitlines()

    new_file_name = cmdline.filename[:-4] + "solution.txt"
    f = open(new_file_name, "w")

    # first line
    for i in range(4):
        f.write(str(assignment[variables[i]]))
    f.write ("\n")

    # second line
    for i in range(4,8):
        f.write(str(assignment[variables[i]]))
    f.write ("\n")

    # third line
    for i in range(8,13):
        f.write(str(assignment[variables[i]]))

# Main function
def main() -> None:
    # Parse command-line arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('filename')
    cmdline = parser.parse_args()

    # Read input file and create initial and goal states
    with open(cmdline.filename, 'r') as file:
        lines = file.read().splitlines()

    # get all variables
    variables = []
    clean_variables = []
```

```
    # line 1:
    for i in range(4):
        variables += [lines[0][i]]
        clean_variables += [lines[0][i]]

    # line 2:
    for i in range(4):
        variables += [lines[1][i]]
        clean_variables += [lines[1][i]]

    # line 3:
    for i in range(5):
        variables += [lines[2][i]]
        clean_variables += [lines[2][i]]

    # make a set of unique variables
    unique_variables = set(variables)

    # get the constraints set up
    constraints = set_up_constraints(variables,
unique_variables)

    # Initialize domains based on variable-specific constraints
    domains = {}

    index = 1
    for var in variables:
        if index == 1 or index == 5:
            domains[var] = list(range(1, 10))
        elif index == 9:
            domains[var] = [1]
        else:
            domains[var] = list(range(10))
        index += 1
    domains["alpha"] = [0, 1]
    domains["beta"] = [0, 1]
    domains["gamma"] = [0, 1]

    assignment = {}
    backtracking_search(variables, unique_variables, domains,
constraints, assignment, clean_variables)

if __name__ == "__main__":
    main()
```