**Project 2: Futoshiki Solver**

**CS-UY 4563**

**By: Tomas David and Mateo Castro**

**Due: May 17th, 2021**

**Instructions**

1. Install python on your console with “Brew” or “pip” on your root directory as such:

“Brew install python”

1. Download the entire folder as it is. Then navigate to the directory “15PuzzleAIProject” as such:

“cd 15PuzzleAIProject”.

1. Then run “python Futoshiki.py”.
2. Please create an empty .txt file in the same directory as the root called “SampleOutput.txt”.
3. You will be prompted to enter an input file name, until a match is found. Press enter once you have written it down in the console. Note: only files in the root of the project directory will be found, otherwise you must include the relative path to this directory. Example “Input1.txt”.
4. The generated output will be written down in “SampleOutput.txt”. Note: this file gets re-written every time the algorithm is run with a new input. If you want to keep a record of the output, make a copy of “SampleOutput.txt” before re-running the code.
5. The input files “Input1.txt”, “Input2.txt”, “Input3.txt”, have been included in the directory for your convenience.

**Source Code**

import copy  
import sys  
  
# initial: Initial Board (2D Array)  
# horiz\_constraints: Horizontal Constraints (2D Array)  
# vert\_constraints: Vertical Constraints (2D Array)  
# variables: All variables are strings in form "xij" (List)  
# domains: Domains for all variables (Dictionary)  
# degrees: Degrees (# of unassigned neighbors) of all variables (Dictionary)  
class CSP:  
 def \_\_init\_\_(self, initial, horiz\_constraints, vert\_constraints):  
 self.initial = initial  
 self.horiz\_constraints = horiz\_constraints  
 self.vert\_constraints = vert\_constraints  
 self.variables = []  
 for i in range(0, 6, 1):  
 for j in range(0, 6, 1):  
 self.variables.append("x" + str(i) + str(j))  
 self.domains = {}  
 for var in self.variables:  
 self.domains[var] = [1, 2, 3, 4, 5, 6]  
 self.degrees = {}  
 for var in self.variables:  
 self.degrees[var] = 0  
  
  
 # Updates neighbors of variable at row,col  
 # Used for forward checking  
 def updateNeighborDomains(self, row, col):  
 num = int(self.initial[row][col])  
  
 # Removes value at i,j from its neighbors domains (AllDiff)  
 # Column neighbors  
 for c in range(0, 6, 1):  
 var = "x" + str(row) + str(c)  
 if num in self.domains[var]:  
 self.domains[var].remove(int(num))  
  
 # Row neighbors  
 for r in range(0, 6, 1):  
 var = "x" + str(r) + str(col)  
 if num in self.domains[var]:  
 self.domains[var].remove(int(num))  
  
 # Removes values > or < from right neighbor  
 if col < 5:  
 neighbor\_var = "x" + str(row) + str(col + 1)  
 if self.horiz\_constraints[row][col] == ">":  
 self.domains[neighbor\_var] = [x for x in self.domains[neighbor\_var] if x < num]  
 elif self.horiz\_constraints[row][col] == "<":  
 self.domains[neighbor\_var] = [x for x in self.domains[neighbor\_var] if x > num]  
  
 # Removes values ^ or v from bottom neighbor  
 if row < 5:  
 neighbor\_var = "x" + str(row + 1) + str(col)  
 if self.vert\_constraints[row][col] == "v":  
 self.domains[neighbor\_var] = [x for x in self.domains[neighbor\_var] if x < num]  
 elif self.vert\_constraints[row][col] == "^":  
 self.domains[neighbor\_var] = [x for x in self.domains[neighbor\_var] if x > num]  
  
  
 # Updates domains of neighbors for initial values  
 def forwardCheck(self):  
 for i in range(0, 6, 1):  
 for j in range(0, 6, 1):  
 if self.initial[i][j] != "0":  
 self.updateNeighborDomains(i, j)  
  
  
 # MRV used for selecting next variable to work on  
 def minimumRemainingValuesHueristic(self, vars):  
 min = 7  
 min\_remaining = []  
  
 # Determine min size of remaining domains  
 for var in vars:  
 if len(self.domains[var]) < min:  
 min = len(self.domains[var])  
  
 # Obtain domains with min size  
 for var in vars:  
 if len(self.domains[var]) == min:  
 min\_remaining.append(var)  
  
 #return the smallest domain  
  
 return min\_remaining  
  
  
 # Determines if the given board is consistent with constraints  
 def isConsistent(self, var, assignment):  
  
 diff = False  
 horiz\_const = False  
 vert\_const = False  
  
 row = int(var[1])  
 col = int(var[2])  
  
 diff = allDiff(row, col, assignment)  
 horiz\_const = self.checkHorizontalConstraint(var, assignment)[0]  
 vert\_const = self.checkVerticalConstraint(var, assignment)[0]  
  
 return diff and horiz\_const and vert\_const  
  
 # Determines if var satisfies horizontal constraints  
 # And the number of unassigned horizontal neighbors  
 def checkHorizontalConstraint(self, var, assignment):  
 row = int(var[1])  
 col = int(var[2])  
  
 val = int(assignment[row][col])  
  
 valid\_right = True  
 valid\_left = True  
  
 horizontal\_constraints = 0  
  
 # Check if there is a inequality to the right  
 if col < 5:  
 neighbor\_val = int(assignment[row][col + 1])  
 if neighbor\_val == 0:  
 valid\_right = True  
 elif self.horiz\_constraints[row][col] == ">" and val < neighbor\_val:  
 valid\_right = False  
 horizontal\_constraints+=1  
 elif self.horiz\_constraints[row][col] == "<" and val > neighbor\_val:  
 valid\_right = False  
 horizontal\_constraints+=1  
  
 # Check if there is a inequality to the left  
 if col > 0:  
 neighbor\_val = int(assignment[row][col - 1])  
 if neighbor\_val == 0:  
 valid\_left = True  
 elif self.horiz\_constraints[row][col - 1] == ">" and val > neighbor\_val:  
 valid\_left = False  
 horizontal\_constraints+=1  
 elif self.horiz\_constraints[row][col - 1] == "<" and val < neighbor\_val:  
 valid\_left = False  
 horizontal\_constraints+=1  
  
 return (valid\_left and valid\_right, horizontal\_constraints)  
  
  
 # Determines if var satisfies vertical constraints  
 # And the number of unassigned vertical neighbors  
 def checkVerticalConstraint(self, var, assignment):  
 row = int(var[1])  
 col = int(var[2])  
  
 val = int(assignment[row][col])  
  
 vertical\_contstraints = 0  
  
 valid\_top = True  
 valid\_bottom = True  
  
 # Check if there is a inequality to the bottom  
 if row < 5:  
 neighbor\_val = int(assignment[row + 1][col])  
 if neighbor\_val == 0:  
 valid\_bottom = True  
 elif self.vert\_constraints[row][col] == "v" and val < neighbor\_val:  
 valid\_bottom = False  
 vertical\_contstraints += 1  
 elif self.vert\_constraints[row][col] == "^" and val > neighbor\_val:  
 valid\_bottom = False  
 vertical\_contstraints += 1  
  
 # Check if there is a inequality to the top  
 if row > 0:  
 neighbor\_val = int(assignment[row - 1][col])  
 if neighbor\_val == 0:  
 valid\_top = True  
 elif self.vert\_constraints[row - 1][col] == "v" and val > neighbor\_val:  
 valid\_top = False  
 vertical\_contstraints += 1  
 elif self.vert\_constraints[row - 1][col] == "^" and val < neighbor\_val:  
 valid\_top = False  
 vertical\_contstraints += 1  
  
 return (valid\_bottom and valid\_top,vertical\_contstraints)  
  
  
# Determines which variables to use next depending on their number of unassigned neighbors  
def degreeHeuristic(vars,board,csp):  
 max\_vars = []  
  
 global\_max = 0  
  
 for var in vars:  
 csp.degrees[var] = getDegree(var, board,csp)  
 max = csp.degrees[var]  
 if global\_max < max:  
 global\_max = max  
  
 for var in vars:  
 if global\_max == csp.degrees[var]:  
 max\_vars.append(var)  
  
 return max\_vars  
  
  
# Gets the number of unassigned neighbors  
# Used in degreeHeuristic  
def getDegree(var, board,csp):  
  
 degree,vertical\_constrains,horizontal\_constrains = 0,0,0  
  
 col = int(var[2])  
 row = int(var[1])  
  
 for c in range(0, 6, 1):  
 if board[row][c] == '0':  
 degree += 1  
  
  
 vertical\_constrains = csp.checkVerticalConstraint(var,board)[1]  
 horizontal\_constrains = csp.checkHorizontalConstraint(var,board)[1]  
  
 return degree+vertical\_constrains+horizontal\_constrains  
  
  
# Checks if all values in a list are different (Except for 0)  
def checkDiff(elems):  
 for elem in elems:  
 #Don't check for 0  
 if(elem == "0"):  
 continue  
  
 # Returns False if a duplicate is found  
 if elems.count(elem) > 1:  
 return False  
  
 # Returns true if all elements are "0" or there isn't a duplicate  
 return True  
  
  
# AllDiff constraint for a row and col  
def allDiff(row, col, board):  
 row\_elems = []  
 col\_elems = []  
  
 for c in range(0, 6, 1):  
 row\_elems.append(board[row][c])  
  
 for r in range(0, 6, 1):  
 col\_elems.append(board[r][col])  
  
 return checkDiff(row\_elems) and checkDiff(col\_elems)  
  
  
# Chooses a variable to work on next  
# Uses MRV and Degree Heuristics  
def selectUnassignedVariable(csp, assignment):  
 vars = []  
  
 for i in range(0, 6, 1):  
 for j in range(0, 6, 1):  
 if assignment[i][j] == "0":  
 vars.append("x" + str(i) + str(j))  
  
 vars = csp.minimumRemainingValuesHueristic(vars)  
  
 #In case of a tie between the min domain use degreeHeuristic:  
 if len(vars) > 1 :  
 vars = degreeHeuristic(vars,assignment,csp)  
 return vars[-1]  
  
  
# Determines if the board is complete  
def isComplete(assignment):  
 for i in range(0, 6, 1):  
 for j in range(0, 6, 1):  
 if assignment[i][j] == "0":  
 return False  
 return True  
  
  
# Backtracking Search  
def backTrackingSearch(csp):  
 initial\_copy = copy.deepcopy(csp.initial)  
 return backtrack(csp, initial\_copy)  
  
  
# Recursive move of backtracking search  
def backtrack(csp, assignment):  
 if isComplete(assignment):  
 return assignment  
  
 var = selectUnassignedVariable(csp, assignment)  
 row = int(var[1])  
 col = int(var[2])  
  
  
 for value in csp.domains[var]:  
 assignment[row][col] = str(value)  
  
 if csp.isConsistent(var, assignment):  
 result = backtrack(csp, assignment)  
 if result != None:  
 return result  
  
 assignment[row][col] = "0"  
  
 return None  
  
  
#---------------------------------Main------------------------------  
initial = []  
horiz\_ineq = []  
vert\_ineq = []  
  
# Ask to initialize the file until found  
while True:  
 print("\nEnter the name of the input, ensuring that you type .txt afterwards.\n\nOutput will be written in SampleOutput.txt. Please check your directory. ")  
 file\_name = input();  
 try:  
 input\_file = open(file\_name)  
 if input\_file:  
 break  
 except IOError:  
 print ("There is no such a file, please try again")  
  
input\_file = open(file\_name)  
  
input\_str = input\_file.read()  
  
# Splits with each new line  
split\_input = input\_str.split('\n')  
  
  
# Open output file for writing  
sys.stdout = open("SampleOutput.txt", "w")  
  
# Clean the output file  
file = open("SampleOutput.txt","r+")  
file.truncate(0)  
file.close()  
  
switch = 0  
curr\_elem = ""  
  
# Splits with each space  
# Appends to initial if switch = False  
# else appends to goal  
for arr in split\_input:  
 if arr == '':  
 switch += 1  
 continue  
  
 if switch == 0:  
 initial.append(arr.split(" "))  
 elif switch == 1:  
 horiz\_ineq.append(arr.split(" "))  
 else:  
 vert\_ineq.append(arr.split(" "))  
  
# Solution for CSP initialized  
problem = CSP(initial, horiz\_ineq, vert\_ineq)  
  
# Apply forward checking before starting search  
problem.forwardCheck()  
  
solution = backTrackingSearch(problem)  
  
# Print solution  
for row in range(0, 6, 1):  
 for col in range(0, 6, 1):  
 print(solution[row][col], end = " ")  
 print()  
  
input\_file.close()  
sys.stdout.close()

**Input Files and Output Files**

**Input File 1**

0 0 0 0 0 0  
3 0 0 0 0 0  
0 3 0 0 6 4  
0 0 0 0 4 0  
0 0 0 0 0 0  
0 0 0 0 1 3  
  
0 0 > 0 0  
0 0 0 0 0  
0 0 0 0 0  
0 0 0 < 0  
0 > 0 0 >  
> 0 0 0 0  
  
v 0 0 0 0 0  
0 0 0 0 0 0  
0 0 0 ^ 0 0  
0 0 0 0 0 ^  
v 0 0 0 0 0

**Output File 1**

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

**Input File 2**

5 0 0 0 0 0  
0 0 0 0 0 0  
0 0 0 0 0 0  
0 2 0 0 0 0  
0 0 5 0 3 0  
3 5 4 0 0 0  
  
0 < 0 0 0  
0 0 0 0 0  
0 0 0 0 0  
0 < 0 < 0  
0 < 0 0 0  
0 0 0 0 <  
  
0 0 0 0 0 0  
^ 0 0 0 0 0  
0 0 v 0 0 v  
v 0 0 0 0 0  
0 0 0 ^ 0 0

**Output File 2**

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

**Input File 3**

6 0 0 0 0 0  
0 0 0 0 0 0  
0 0 0 0 0 0  
0 0 0 0 0 0  
0 0 5 0 0 0  
0 0 0 0 0 3  
  
0 < 0 0 <  
0 0 < < 0  
> 0 0 > 0  
0 0 < 0 0  
0 0 0 < 0  
0 0 0 < 0  
  
0 0 0 0 0 0  
^ ^ 0 0 0 ^  
0 0 0 0 0 0  
v 0 0 0 v 0  
^ 0 v 0 0 0

**Output File 3**

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