Herman Lin, Mahika Jain
CS 4613 - Artificial Intelligence
Professor Wong
Final Project

GitHub Link: https://github.com/HermanLin/Futoshiki-Project.git

Instructions for Running the Program:

1. If the files are in *.txt format, convert the files into *.py files
2. Go to the directory the files are located
3. Run: `python Main.py` or `python3 Main.py` in your terminal

Output files from Input Test Files:

Output1.txt

```
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
```

Output2.txt

```
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
```

Output3.txt

```
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
```

Source Code: Main.py

```
from CSP import CSP
from Backtracking import BacktrackingSearch

def processFile(filename, outputFilename):
    f = open(filename, 'r')
    # read 6 lines and remove trailing newlines for initial game board
    initialBoard = [next(f).rstrip().split(' ') for line in range(6)]
    for i in range(6):
        for j in range(6):
            initialBoard[i][j] = int(initialBoard[i][j])
    next(f) # skip blank line
    # read 6 lines and remove trailing newlines for horizontal constraints
    horzConstraints = [next(f).rstrip().split(' ') for line in range(6)]
    next(f) # skip blank line
    # read 6 lines and remove trailing newlines for vertical contraints
    vertConstraints = [next(f).rstrip().split(' ') for line in range(5)]
    f.close()

    problem = CSP(initialBoard, horzConstraints, vertConstraints)
    print("===== Initial Board =====")
    problem.printBoard()

    BTS = BacktrackingSearch(problem)
    solution = BTS.backtrack(problem)

    if solution:
        print("======= SOLUTION ========")
        for row in solution: print(row)
        print()
    else: print("NO SOLUTION FOUND\n")
    outputFile(outputFilename, solution)

def outputFile(outputFilename, board):
    file = open(outputFilename, 'w')
    for row in board:
        for cell in row:
            file.write(str(cell) + " ")
        file.write("\n")
    file.close()

def main():
    processFile("Input1.txt", "Output1.txt")
    processFile("Input2.txt", "Output2.txt")
    processFile("Input3.txt", "Output3.txt")

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

Source Code: CSP.py

```
class CSP:

    def __init__(self, board, hc, vc):
        self.board = board
        self.domains = []
        self.hc = hc
        self.vc = vc

        self.initDomains()
        self.initConstraintCheck()
        self.forwardCheck()

    # initialize the domains of the board
    def initDomains(self):
        for i in range(len(self.board)):
            row_domains = [[1,2,3,4,5,6] for j in range(6)]
            self.domains.append(row_domains)

    # restrict domains based on inequalities once
    def initConstraintCheck(self):
        for row in range(len(self.board)):
            for col in range(len(self.board[row])):
                # horizontal constraint check
                if col - 1 >= 0: # middle columns 1-5
                    if self.hc[row][col - 1] == '<':
                        self.domains[row][col - 1] = self.domains[row][col-1][:-1]
                        self.domains[row][col] = self.domains[row][col][1:]
                    elif self.hc[row][col - 1] == '>':
                        self.domains[row][col - 1] = self.domains[row][col-1][1:]
                        self.domains[row][col] = self.domains[row][col][:-1]
                # vertical constraint check
                if row - 1 >= 0: # middle row 1-5
                    if self.vc[row - 1][col] == '^':
                        self.domains[row-1][col] = self.domains[row-1][col][:-1]
                        self.domains[row][col] = self.domains[row][col][1:]
                    elif self.vc[row - 1][col] == 'v':
                        self.domains[row-1][col] = self.domains[row-1][col][1:]
                        self.domains[row][col] = self.domains[row][col][:-1]


    # perform forwardChecking on a board
    def forwardCheck(self):
        updateFailed = False
        for row in range(len(self.board)):
            for col in range(len(self.board[row])):
                if updateFailed: return False
                if self.board[row][col] != 0: # the board has a number
                    # set respective domain to val in cell
```

```
                    self.domains[row][col] = [self.board[row][col]]
                    if not self.updateNeighbors(row, col):
                        updateFailed = True
        return True


    def updateNeighbors(self, row, col):
        value = self.board[row][col]
        # update row neighbors
        for i in range(len(self.domains)):
            if i == col: pass
            else:
                try: self.domains[row][i].remove(value)
                except ValueError: pass
        # update col neighbors
        for i in range(len(self.domains)):
            if i == row: pass
            else:
                try: self.domains[i][col].remove(value)
                except ValueError: pass

        # check constraints for the (row, col)
        return self.checkConstraints(row, col)


    def checkConstraints(self, row, col): # check value is legal with constraints
applied
        value = self.board[row][col]
        new_domain = []

        if col - 1 >= 0: # middle columns 1-5
            if self.hc[row][col - 1] == '<':
                new_domain = [x for x in self.domains[row][col - 1] if x < value]
                if not new_domain: return False
                self.domains[row][col - 1] = new_domain
            elif self.hc[row][col - 1] == '>':
                new_domain = [x for x in self.domains[row][col - 1] if x > value]
                if not new_domain: return False
                self.domains[row][col - 1] = new_domain

        if col <= 4: # middle columns 0-4
            if self.hc[row][col] == '<':
                new_domain = [x for x in self.domains[row][col + 1] if x > value]
                if not new_domain: return False
                self.domains[row][col + 1] = new_domain
            elif self.hc[row][col] == '>':
                new_domain = [x for x in self.domains[row][col + 1] if x < value]
                if not new_domain: return False
                self.domains[row][col + 1] = new_domain
```

```
        if row - 1 >= 0:
            if self.vc[row - 1][col] == '^':
                new_domain = [x for x in self.domains[row - 1][col] if x < value]
                if not new_domain: return False
                self.domains[row - 1][col] = new_domain
            elif self.vc[row - 1][col] == 'v':
                new_domain = [x for x in self.domains[row - 1][col] if x > value]
                if not new_domain: return False
                self.domains[row - 1][col] = new_domain

        if row <= 4:
            if self.vc[row][col] == '^':
                new_domain = [x for x in self.domains[row + 1][col] if x > value]
                if not new_domain: return False
                self.domains[row + 1][col] = new_domain
            elif self.vc[row][col] == 'v':
                new_domain = [x for x in self.domains[row + 1][col] if x < value]
                if not new_domain: return False
                self.domains[row + 1][col] = new_domain

        return True

    # minimum remaining values heuristic
    def MRV(self):
        minimum = 6
        min_pos = []
        for row in range(len(self.domains)):
            for col in range(len(self.domains[row])):
                domain_len = len(self.domains[row][col])
                if domain_len == 1:
                    if self.board[row][col] != 0:
                        continue
                if domain_len < minimum:
                    minimum = domain_len # new len found
                    min_pos = [] # reset minimum position array
                    min_pos.append((row, col))
                elif domain_len == minimum:
                    min_pos.append((row, col))
        return min_pos

    # degrees heuristic
    def degrees(self, row, col):
        remaining_neighbors = 0

        # count how many row neighbors are unassigned
        remaining_neighbors += self.board[row].count(0)
        # count how many col neighbors are unassigned
        col_neighbors = [self.board[r][col] for r in range(len(self.board))]
```

```
        remaining_neighbors += col_neighbors.count(0)

        return remaining_neighbors

    # heuristic function for choosing next unassigned cell to backtrack
    def heuristic(self):
        min_pos = self.MRV()
        degrees = []
        for pos in min_pos:
            row, col = pos[0], pos[1]
            degrees.append(self.degrees(row, col))

        sorted_min_pos = [x for _, x in sorted(zip(degrees, min_pos))]
        # need to check according to highest degree first
        sorted_min_pos.reverse()

        return sorted_min_pos

    # check if an assignment is consistent within the respective row and col
    def checkAllDiff(self, row, col):
        value = self.board[row][col]
        # check for all different in row
        if self.board[row].count(value) > 1: return False
        # check for all different in col
        column = [self.board[i][col] for i in range(6)]
        if column.count(value) > 1: return False

        return True # all diff!

    # check if an assignment satisfies nearby inequalities (if they exist)
    def checkInequal(self, row, col):
        value = self.board[row][col]

        if col - 1 >= 0: # middle columns 1-5
            neighbor_value = self.board[row][col - 1]
            if self.hc[row][col - 1] == '<':
                if value < neighbor_value and neighbor_value: return False
            elif self.hc[row][col - 1] == '>':
                if value > neighbor_value and neighbor_value: return False

        if col <= 4: # middle columns 0-4
            neighbor_value = self.board[row][col + 1]
            if self.hc[row][col] == '<':
                if value > neighbor_value and neighbor_value: return False
            elif self.hc[row][col] == '>':
                if value < neighbor_value and neighbor_value: return False

        if row - 1 >= 0:
            neighbor_value = self.board[row - 1][col]
```

```
            if self.vc[row - 1][col] == '^':
                if value < neighbor_value and neighbor_value: return False
            elif self.vc[row - 1][col] == 'v':
                if value > neighbor_value and neighbor_value: return False

        if row <= 4:
            neighbor_value = self.board[row + 1][col]
            if self.vc[row][col] == '^':
                if value > neighbor_value and neighbor_value: return False
            elif self.vc[row][col] == 'v':
                if value < neighbor_value and neighbor_value: return False

        return True

    # check if an assignment is consistent with the board and constraints
    def isConsistent(self, row, col):
        all_diff = self.checkAllDiff(row, col)
        inequal = self.checkInequal(row, col)
        return all_diff and inequal

    # check if a board is complete
    def isComplete(self):
        for row in self.board:
            for cell in row:
                if cell == 0:
                    return False
        return True

    def printBoard(self):
        for row in self.board:
            print(row)

    def printDomains(self):
        for domain in self.domains:
            print(domain)
```

Source Code: Backtracking.py

```
from CSP import CSP
import copy

class BacktrackingSearch:

    def __init__(self, csp):
        self.csp = csp

    def backtrack(self, csp):
        if csp.isComplete():
            return csp.board
        # select an unassigned variable to update
        sorted_min_pos = csp.heuristic()

        for pos in sorted_min_pos:
            row, col = pos[0], pos[1]

            # get the domain associated with the (row, col)

            domain = csp.domains[row][col]
            # in order to perform inferencing, save old state
            oldDomains = copy.deepcopy(csp.domains)

            for value in domain:
                # update board with value and check consistency
                csp.board[row][col] = value

                if csp.isConsistent(row, col):
                    # perform forward check
                    if csp.forwardCheck():
                        result = self.backtrack(csp)
                        if result != None: return csp.board

                    # if the assignment is not consistent or fails, reset
                    csp.board[row][col] = 0
                # reset domains
                csp.domains = oldDomains
            return None
        return None
```