Andrew Asseily & Leisha Murthy (aa7342 , lm4684)

Artificial Intelligence

12/18/21

Futoshiki Solver Project 2

Instructions on how to run our program:

1. Open the program and run main.py in the AI-Project2 folder (python3 main.py)
2. Input which test file you'd like to run
3. Search for output files within the same folder

Source Code:

FutoshikiUtility.py

```
class FutoshikUtility:

    # initialize value in the constructor
    def __init__(self, initial_game_state, h_constraints, v_constraints,
heuristic_remainder):
        self.initial_game_state = initial_game_state
        self.h_constraints = h_constraints
        self.v_constraints = v_constraints
        self.heuristic_remainder = heuristic_remainder

    # find minimum remainder from game board
    def min_remainder(self, game_board):
        used_tuples = []
        min = 5
        max = 5
        len_heuristic = len(self.heuristic_remainder)
        index = 0
        used_tuples = self.update_tuples(game_board, index, len_heuristic, max, min,
used_tuples)
        return self.get_cell(game_board, used_tuples)

    # If tiebreaker is not required , return min cell from game board
    def get_cell(self, game_board, used_tuples):
        if len(used_tuples) != 1:
```

```
            return self.tiebreaker(game_board, used_tuples)
        else:
            return used_tuples[0]

    # update tuples value based on heuristic_check
    def update_tuples(self, game_board, index, len_heuristic, max, min, used_tuples):
        while index < len_heuristic:
            len_heuristic_index = len(self.heuristic_remainder[index])
            index2 = 0
            used_tuples = self.heuristic_check(game_board, index, index2,
len_heuristic_index, max, min, used_tuples)

            index += 1
        return used_tuples

    # do heuristic_check based on tuple value
    def heuristic_check(self, game_board, index, index2, len_heuristic_index, max, min,
used_tuples):
        while index2 < len_heuristic_index:
            if game_board[index][index2] != "0":
                index2 += 1
                continue
            min_len = len(self.heuristic_remainder[index][index2])
            current_val = max - min_len
            used_tuples = self.tuple_operation(current_val, index, index2, min,
min_len, used_tuples)

            index2 += 1
        return used_tuples

    # append tuples based on cell index
    def tuple_operation(self, current_val, index, index2, min, min_len, used_tuples):
        if current_val == min:
            used_tuples.append((index, index2))

        elif current_val < min:
            min = min_len
            used_tuples = [(index, index2)]
        return used_tuples

    # check heuristic_remainder and update game board with the latest value
    def update_heuristic_remainder(self, game_board, r, c):
```

```
        board_value = game_board[r][c]
        board_len = len(game_board)
        index = 0
        self.iterate_heuristic_remainder(board_len, board_value, game_board, index, r)
        self.update_game_board(board_value, c, game_board, r)

    # check if current selection is valid and based on that update game board
    def update_game_board(self, board_value, c, game_board, r):
        board_len = len(game_board[r])
        index = 0
        while index < board_len:
            if abs(c - index) > 1:
                self.heuristic_remainder[index][c][board_value] = 0
            else:
                index2 = 1
                while index2 < 6:
                    if not self.checkCurrentSelection(game_board, index, c, index2):
                        self.heuristic_remainder[index][c][index2] = 0
                    index2 += 1
            index += 1

    # find the abs value of row with current index and update heuristic_remainder
    def iterate_heuristic_remainder(self, board_len, board_value, game_board, index,
r):
        while index < board_len:
            # If further away than 1
            abs_value = abs(r - index)
            if abs_value <= 1:
                val = 1
                while val < 6:
                    if not self.checkCurrentSelection(game_board, r, index, val):
                        self.heuristic_remainder[r][index][val] = 0
                    val += 1
            else:
                self.heuristic_remainder[r][index][board_value] = 0
            index += 1

    # find degree heuristic to tie-break
    def tiebreaker(self, game_board, used_tuples):
        graph_deg = {}
        tiebreaker_box = None
        self.set_graph_deg(game_board, graph_deg, used_tuples)
```

```
        max_degree = 0
        tiebreaker_box = self.tiebreaker_box(graph_deg, max_degree, tiebreaker_box)
        return tiebreaker_box

    # return tiebreaker_box
    def tiebreaker_box(self, graph_deg, max_degree, tiebreaker_box):
        for cell, unvisited_neighbors in graph_deg.items():
            if unvisited_neighbors > max_degree:
                tiebreaker_box = cell
                max_degree = unvisited_neighbors
        return tiebreaker_box

    # find number of unassigned_neighbors and return the degree
    def set_graph_deg(self, game_board, graph_deg, used_tuples):
        for t1, t2 in used_tuples:
            unvisited_neighbors = 0
            game_board_len = len(game_board)
            index = 0
            unvisited_neighbors = self.get_unvisited_neighbors_count(game_board,
game_board_len, index, t2,

unvisited_neighbors)
            game_board_len_t2 = len(game_board[t2])
            index2 = 0
            self.get_unvisited_neighbors_count(game_board, game_board_len_t2, index2,
index2,
                                               unvisited_neighbors)
            graph_deg[(t1, t2)] = unvisited_neighbors - 1

    # get the unvisited_neighbors_count
    def get_unvisited_neighbors_count(self, game_board, game_board_len, index, t2,
unvisited_neighbors):
        while index < game_board_len:
            if game_board[index][t2] == "0":
                unvisited_neighbors += 1
            index += 1
        return unvisited_neighbors

    # check the number with all constraint, if all condition satisfied return True, if
any constraint break
    # return false
    def checkCurrentSelection(self, game_board, r, c, value):
```

```
        board_length = len(game_board)
        index = 0
        while index < board_length:
            if c != index and game_board[r][index] == str(value):
                return False
            index += 1

        game_board_row_len = len(game_board[r])
        index = 0
        while index < game_board_row_len:
            if r != index and game_board[index][c] == str(value):
                return False
            index += 1

        if r >= 0 and r != game_board_row_len - 1 and int(game_board[r + 1][c]) != 0:
            if "v" == self.v_constraints[r][c]:
                if int(game_board[r + 1][c]) > value:
                    return False
            if "^" == self.v_constraints[r][c]:
                if int(game_board[r + 1][c]) < value:
                    return False

        if c >= 0 and c != game_board_row_len - 1 and int(game_board[r][c + 1]) != 0:
            if ">" == self.h_constraints[r][c]:
                if int(game_board[r][c + 1]) > value:
                    return False

            if "<" == self.h_constraints[r][c]:
                if int(game_board[r][c + 1]) < value:
                    return False

        if r <= board_length - 1 and r != 0 and int(game_board[r - 1][c]) != 0:
            if "v" == self.v_constraints[r - 1][c]:
                if int(game_board[r - 1][c]) < value:
                    return False

            if "^" == self.v_constraints[r - 1][c]:
                if int(game_board[r - 1][c]) > value:
                    return False

        if c <= game_board_row_len - 1 and c != 0 and int(game_board[r][c - 1]) != 0:
```

```
        if ">" == self.h_constraints[r][c - 1]:
            if int(game_board[r][c - 1]) < value:
                return False

        if "<" == self.h_constraints[r][c - 1]:
            if int(game_board[r][c - 1]) > value:
                return False

    return True

# start checking each cell and put correct value
def execute_game(self, game_board):
    flag = True
    flag = self.checkZeroInBoard(flag, game_board)
    if flag:
        return True
    index = 1
    available_set = self.min_remainder(game_board)
    return self.check_status(available_set, game_board, index)

# if current solution is satisfied with all constraint set the value else 0
def check_status(self, available_set, game_board, index):
    while index < 6:
        row = available_set[0]
        col = available_set[1]
        current_status = self.checkCurrentSelection(game_board, row, col, index)
        if current_status:
            game_board[row][col] = str(index)
            self.update_heuristic_remainder(game_board, row, col)
            is_solved = self.execute_game(game_board)
            if is_solved:
                return True
            else:
                game_board[row][col] = "0"
        index += 1
    return False

# check if any 0 exist, else all cell is filled with correct value
def checkZeroInBoard(self, flag, game_board):
    for rows in game_board:
        for r in rows:
```

```
                if r == "0":
                    flag = False
                    break
        return flag
```

FileOperation.py

```
import re


class FileOperation:
    # constructor
    def __init__(self, fileName):
        self.fileName = fileName
        self.initial_game_state = [["0"] * 5 for x in range(5)]
        self.h_constraints = [["0"] * 4 for x in range(5)]
        self.v_constraints = [["0"] * 5 for x in range(4)]

    # return file number from file name
    def get_numbers_from_filename(self, filename):
        return re.search(r'\d+', filename).group(0)

    # write final valid solution to a text file
    def write_valid_solution(self, output):
        f_number = self.get_numbers_from_filename(self.fileName)
        solution = self.build_final_solution(output)
        output_file_Name = "Output" + str(f_number) + ".txt"
        f = open(output_file_Name, "w+")
        f.write(solution)
        f.close()

    # iterate over each row and column
    def build_final_solution(self, output):
        solution = ""
        len1 = len(output)
        index = 0
        while index < len1:
            index2 = 0
```

```
            len2 = len(output[index])
            while index2 < len2:
                solution += output[index][index2] + " "
                index2 += 1
            solution = solution.rstrip()
            solution += "\n"
            index += 1
        return solution

    # write invalid solution to a text file
    def write_invalid_solution(self):
        f_number = self.get_numbers_from_filename(self.fileName)
        output_file_Name = "Output" + str(f_number) + ".txt"
        f = open(output_file_Name, "w+")
        f.write("No Solution")
        f.close()

    # read from text file and build constraint
    def read_file(self):
        x = []
        f = open(self.fileName, "r")
        file_value = f.read()

        # Remove spaces from data
        file_value = self.remove_space(file_value, x)
        input_sec = file_value.split("\n\n")
        self.build_game_state(input_sec)
        # build horizontal_conditions
        self.build_h_constraints(input_sec)
        # build vertical_conditions
        self.build_v_constraints(input_sec)
        return self.initial_game_state, self.v_constraints, self.h_constraints

    # remove space
    def remove_space(self, file_value, x):
        file_len = len(file_value.split("\n"))
        index = 0
        while index < file_len:
            val = file_value.split("\n")[index].rstrip()
            x.append(val)
            index += 1
        file_value = "\n".join(x)
```

```
        return file_value

    # build game state from input
    def build_game_state(self, input_sec):
        game_b_r = input_sec[0].split("\n")
        index = 0
        while index < 5:
            game_b_c = game_b_r[index].split(" ")
            index2 = 0
            while index2 < 5:
                self.initial_game_state[index][index2] = game_b_c[index2]
                index2 += 1
            index += 1

    # build horizontal_conditions
    def build_h_constraints(self, input_sec):
        hc_row = input_sec[1].split("\n")
        index = 0
        while index < 5:
            hc_col = hc_row[index].split(" ")
            index2 = 0
            while index2 < 4:
                self.h_constraints[index][index2] = hc_col[index2]
                index2 += 1
            index += 1

    # build vertical_conditions
    def build_v_constraints(self, input_sec):
        vc_row = input_sec[2].split("\n")
        index = 0
        while index < 4:
            vc_col = vc_row[index].split(" ")
            index2 = 0
            while index2 < 5:
                self.v_constraints[index][index2] = vc_col[index2]
                index2 += 1
            index += 1
```

main.py

```
from FileOperation import FileOperation
from FutoshikiUtility import FutoshikUtility

heuristic_remainder = [[0] * 5 for x in range(5)]


# initialize each cell of heuristic_remainder
def set_min_rem_val_heuristic():
    heuristic_remainder_len = len(heuristic_remainder)
    index = 0
    while index < heuristic_remainder_len:
        h_l = len(heuristic_remainder[index])
        index2 = 0
        while index2 < h_l:
            heuristic_remainder[index][index2] = {}
            index2 += 1
        index += 1


def main():
    # accept input file from user
    input_file = input("enter file name: ")
    # create instance of FileOperation
    fileOps = FileOperation(input_file)
    # read from input file and initialize game_state, v_constraints and h_constraints
    initial_game_state, v_constraints, h_constraints = fileOps.read_file()

    set_min_rem_val_heuristic()

    # create instance of FutoshikUtility and pass call the constraint
    utility = FutoshikUtility(initial_game_state, h_constraints, v_constraints,
heuristic_remainder)
    valid = utility.execute_game(initial_game_state)
    # if game doesn't have valid solution, display user defined message
    if not valid:
        fileOps.write_invalid_solution()
    else:
        fileOps.write_valid_solution(initial_game_state)

# start of program
if __name__ == "__main__":
```

```
main()
```

Outputs:

Output1.txt

****

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

Output2.txt

****

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

Output3.txt

****

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