Mark Bekker

Mb6386

14-tile Puzzle Instructions

Instructions:

To run the code, simply download the 14-puzzles.py file and under the main() function assign the name of the input file you wish to run to the variable called FILE_NAME. Then just run the code and an output file will be generated in the same directory as the py file and input files.

Output Text Files:

Output1.txt:

1 2 3 4

5 0 6 7

8 9 0 10

11 12 13 14

1 2 4 0

8 5 3 7

11 9 6 10

0 12 13 14

6

193

L1 D1 D1 U2 U2 R2

6 6 6 6 6 6

Output2.txt:

1 5 3 13

8 0 6 4

0 10 7 9

11 14 2 12

1 3 4 13

8 5 7 9

10 0 6 12

11 14 0 2

12

12150

R2 R1 R1 D1 D1 L1 U2 U2 R2 D2 D2 L2

9 9 10 11 11 11 11 12 12 12 12 12 12

Output3.txt:

9 13 7 4

12 3 0 1

2 0 5 6

14 10 11 8

9 3 13 4

2 7 1 0

10 12 0 5

14 11 8 6

14

10534

D2 R2 R2 U2 L2 U1 L1 D1 L1 D1 R1 U1 R1 R1

11 12 12 12 12 12 13 13 13 14 14 14 14 14 14

Source Code:

```
#Object that stores each individual node of the puzzle along with its current state or data, depth level, fvalue, previous node, and previous move
class Node:
    def __init__(self, data, level, fvalue, prev, prev_move):
        """ Initialize the node with the data, level of the node, and the calculated fvalue """
        self.data = data
        self.level = level
        self.fvalue = fvalue
        self.prev = prev
        self.prev_move = prev_move

    #Generate child nodes from the current node
    def generate_child(self):

        #Coordinates of first blank tile
        x,y = self.find_blank_tile(self.data, 'A')
        #Coordinates of second blank tile
        a,b = self.find_blank_tile(self.data, 'B')

        #Potential moves for first blank tile
        first_blank_val_list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]

        # Potential moves for second blank tile
        second_blank_val_list = [[a, b - 1], [a, b + 1], [a - 1, b], [a + 1, b]]

        #Generate valid children and add them to the children list for the first blank tile
        children = []
        for move_type, val in enumerate(first_blank_val_list):
            child = self.shift(self.data, x, y, val[0], val[1])
            if child is not None:

                #Determine the type of move performed (left, right, up, down)
                if move_type == 0:
                    move = "L1"
                elif move_type == 1:
                    move = "R1"
                elif move_type == 2:
                    move = "U1"
                elif move_type == 3:
                    move = "D1"

                child_node = Node(child, self.level+1, 0, self, move)
                children.append((child_node, move))

        #Generate valid children and add them to the children list for the second blank tile
        for move_type, val in enumerate(second_blank_val_list):
```

```
            child = self.shift(self.data, a, b, val[0], val[1])
            if child is not None:

                # Determine the type of move performed (left, right, up, down)
                if move_type == 0:
                    move = "L2"
                elif move_type == 1:
                    move = "R2"
                elif move_type == 2:
                    move = "U2"
                elif move_type == 3:
                    move = "D2"

                child_node = Node(child, self.level + 1, 0, self, move)
                children.append((child_node, move))

        return children

    #Shift the blank space in the specified direction and return the new board or return None if the
requested shift is invalid (goes off board)
    def shift(self, board, x1, y1, x2, y2):
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_board = self.create_copy(board)
            temp = temp_board[x2][y2]
            temp_board[x2][y2] = temp_board[x1][y1]
            temp_board[x1][y1] = temp
            return temp_board

        return None

    def create_copy(self, board):
        """ Copy function to create a similar matrix of the given node"""
        copy_puzzle = []
        for row in board:
            t = []
            for char in row:
                t.append(char)
            copy_puzzle.append(t)
        return copy_puzzle

    #Returns the indices of the requested blank tile (either tile A or tile B)
    def find_blank_tile(self, board, blank):
        """ Specifically used to find the position of the blank space """
        for i in range(0, len(self.data)):
            for j in range(0, len(self.data)):
                if board[i][j] == blank:
                    return (i,j)
```

```
#Object that stores the puzzle as it is being solved along with the size of the board (default at 4x4, the
nodes generated to solve the puzzle, the frontier, and the explored set
class Puzzle:
    def __init__(self, n=4):
        """ Initialize the puzzle size by the specified size, frontier, and explored sets to empty """
        self.n = n
        self.nodes_generated = 0
        self.frontier = []
        self.explored = []

    def read_file(self, filename):
        file = open(filename, "r")
        line = file.readline()
        init_state = []
        goal_state = []
        # Gather initial state puzzle
        for i in range(4):
            row = line.split()
            init_state.append(row)
            line = file.readline()
        line = file.readline()
        # Gather goal state puzzle
        for i in range(4):
            row = line.split()
            goal_state.append(row)
            line = file.readline()

        return (init_state, goal_state)

    def create_output_file(self, filename):
        output_name = filename.split('.')[0]
        output_name +=' Results.txt'
        output_file = open(output_name, 'w+')
        return (output_file, output_name)

    #Replace the 0s in the tiles with either A or B to properly distinguish between the two tiles
    def create_unique_tiles(self, board):
        counter = 1
        new_puzzle = []
        for i in board:
            row = []
            for j in i:
                if j == '0':
                    if counter == 1:
                        row.append('A')
                    elif counter == 2:
                        row.append('B')
                    counter += 1
```

```
                else:
                    row.append(j)
            new_puzzle.append(row)
        return new_puzzle

    #Calculates the f value: f(x) = h(x) + g(x) where h represents the heuristic and g represents the depth level
    def f(self, start, goal):
        return (self.h(start.data, goal) + start.level)

    #Calculates the manhattan distance between the start and goal orientations of the puzzle
    def h(self, start, goal):
        manhattan = 0
        for i in range(0, self.n):
            for j in range(0, self.n):
                if start[i][j] != 'A' and start[i][j] != 'B':
                    if start[i][j] != goal[i][j]:
                        manhattan += 1
        return manhattan

    def display(self, node):
        for i in node.data:
            for j in i:
                print(j, end=" ")
            print("")

    def write_results(self, node, output_file, output_name):
        for row in node.data:
            for char in row:
                if char != 'A' and char != 'B':
                    output_file.write(char)
                else:
                    output_file.write('0')
                output_file.write(' ')
            output_file.write('\n')
        output_file.write('\n')
        output_file.write(f'{node.level}\n')
        output_file.write(f'{self.nodes_generated}\n')

        #Backtrack the nodes from end to start (and then reverse them for proper order) and get the moves from state to state, should be as many moves as depth level
        moves_in_reverse = []
        f_values_in_reverse = []
        while node is not None:
            f_values_in_reverse.append(node.fvalue)
            moves_in_reverse.append(node.prev_move)
            node = node.prev
        moves_in_order = moves_in_reverse[::-1]
```

```python
        moves_in_order = moves_in_order[1:]

        # Backtrack the nodes from end to start (and then reverse them for proper order) and get the f values of each state, should be as many as depth level + 1 for root node
        f_values_in_order = f_values_in_reverse[::-1]
        for move in moves_in_order:
            output_file.write(f'{move} ')
        output_file.write('\n')
        for fval in f_values_in_order:
            output_file.write(f'{fval} ')
        output_file.write('\n')

        #Alert user that results txt file has been created
        print(f'<--------------------------------------------------------------------------------->\nThe results to your puzzle have been generated under the file name: {output_name}.\n<-------------------------------------------------------------------------------->')

    #Solve the puzzle and generate output file displaying all results in the following order: Start state, Goal State, depth level, Nodes generated, Moves from start to finish, fvalues of nodes from start to finish
    def solve(self, FILE_NAME):

        #Read initial state and goal states from the specified file
        init, goal = self.read_file(FILE_NAME)

        #Create an output file and write the initial state into it
        output, output_name = self.create_output_file(FILE_NAME)
        for row in init:
            for char in row:
                output.write(char)
                output.write(' ')
            output.write('\n')
        output.write('\n')

        #Replace 0s with A and B to distinguish tiles
        init = self.create_unique_tiles(init)

        #Create start node and put it in the frontier
        start = Node(init, 0, 0, None, None)
        start.fvalue = self.f(start, goal)
        self.frontier.append((start, ""))
        self.nodes_generated+=1

        not_solved = True
        while not_solved:
            # Select node with the lowest f value for expansion
            cur = self.frontier[0][0]

            # Add current node to explored set so we do not arrive at it again
```

```
            self.explored.append(cur.data)

            #If the manhattan distance between current and goal node is 0 we have reached the goal node
(h(n) = 0) and can write results into the output file
            if self.h(cur.data, goal) == 0:
                self.write_results(cur, output, output_name)
                not_solved = False

            #Go through children of expanded node and add them to the frontier if they have not been
explored yet
            for (i, move) in cur.generate_child():
                i.fvalue = self.f(i, goal)
                if i.data not in self.explored:
                    self.frontier.append((i, move))
                    self.nodes_generated += 1

            #Remove expanded node from the frontier
            del self.frontier[0]

            #Sort the frontier to get ascending f values, the first value in the frontier should have the lowest f
value and therefore be first expanded
            self.frontier.sort(key=lambda x: x[0].fvalue, reverse=False)

def main():
    #Change input file name here:
    FILE_NAME = "Input1.txt"

    #Instantiate puzzle object
    puzzle = Puzzle()

    #Solve puzzle
    puzzle.solve(FILE_NAME)

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