Mark Deng

Instruction:

The code should run right off the bat.

The first input asks for the name of the input text file.

The second input asks for either h1 or h2. Type h1 or h2 for the desired heuristic operation.

The third input asks for the name of the output text file's name. The name input will be followed by h1 or h2 depending on the which heuristic used.

Source Code:

```
'''
The node class is made to instantiate each 2d list table with multiple
variables
'''
class node:
    def __init__(self, data=None, depth=None, move=None, f=None):
        self.data = data  # 2d matrix representation of the board
        if self.data != None:
            self.blank = blank(data)  # a tuple labeling the index of the
blank in the matrix
        else:
            self.blank = None
        self.prev = None
        self.depth = depth  # depth of the node in the search tree
        self.move = move  # the directional move that created this node
        self.f = f       # stores the f(x) value of the node
        self.string = '' # the string representation is used for hashing
        if(self.data != None):
            for i in range(3):
                for j in range(3):
                    self.string += str(data[i][j])

'''
    Searches for the position of the blank
    return as a tuple (x,y)
'''
def blank(matrix):
    for i in range(3):
        for j in range(3):
            if matrix[i][j] == '0':
                return i,j   # return a tuple with x,y position
    raise Exception("Blank space not found in data.")

def switch(matrix, position1, position2):
    new_matrix = [[0,0,0],[0,0,0],[0,0,0]]
    for i in range(3):
        for j in range(3):
            new_matrix[i][j] = matrix[i][j]
    new_matrix[position1[0]][position1[1]] ,
new_matrix[position2[0]][position2[1]] =
```

```
new_matrix[position2[0]][position2[1]] ,
new_matrix[position1[0]][position1[1]]
    return new_matrix
'''
Heuristic function h1 use the sum of Manhattan distances
of the tiles from goal state
input: 2d array of current state and goal
'''
def h1(cur, goal):
    sum = 0
    for i in range(3):
        for j in range(3):
            actual = goal[i][j]
            if actual == '0':
                continue
            for x in range(3):
                for y in range(3):
                    if cur[x][y] == actual:
                        sum += Manhattan((i, j), (x, y))
                        break
    return sum
'''
Heuristic function h2 use Nilsson's sequence score
input: 2d array of current state and goal
The matchups of each cell for both current state and goal state are tabulated
ahead for checks
'''
def h2(cur, goal):
    manhattan = h1(cur, goal)
    match_goal = {
        goal[0][0]: goal[0][1],
        goal[0][1]: goal[0][2],
        goal[0][2]: goal[1][2],
        goal[1][2]: goal[2][2],
        goal[2][2]: goal[2][1],
        goal[2][1]: goal[2][0],
        goal[2][0]: goal[1][0],
        goal[1][0]: goal[0][0],
        goal[1][1]: goal[1][1]
    }
    match_cur = {
        cur[0][0]: cur[0][1],
        cur[0][1]: cur[0][2],
        cur[0][2]: cur[1][2],
        cur[1][2]: cur[2][2],
        cur[2][2]: cur[2][1],
        cur[2][1]: cur[2][0],
        cur[2][0]: cur[1][0],
        cur[1][0]: cur[0][0],
        cur[1][1]: cur[1][1]
    }
    count = 0
    # check center first
    if cur[1][1] != goal[1][1]:
        count += 1
    # check clockwise
    for num in clockwise(cur):
```

```
        if num == '0':  # skip over the blank space
            pass
        elif match_cur[num] != match_goal[num]:
            count += 2
    sum = manhattan + 3 * count
    return sum

'''
An generator for returning each cell of the 2d list in clockwise order
'''
def clockwise(matrix):
    for cell in matrix[0]:
        yield cell
    yield matrix[1][2]
    for index in range(2,-1, -1):
        yield matrix[2][index]
    yield matrix[1][0]

'''
Calculate the Manhattan distance of two coordinates
input: tuples (i,j) and (x,y)
output: integer distance
'''
def Manhattan(set1, set2):
    return abs(set1[0] - set2[0]) + abs(set1[1] - set2[1])

'''
Expand the current node state to at most four new states in the four
directions
output a list of new nodes found that expands from the current node
'''
def expand(curnode, heuristic, goal):
    list = []
    x, y = curnode.blank
    if x < 2:  # down
        newState = switch(curnode.data, curnode.blank, (x+1, y))
        list.append(node(newState, curnode.depth+1, 'D', curnode.depth+1 +
heuristic(newState, goal)))
    if y < 2:  # right
        newState = switch(curnode.data, curnode.blank, (x, y+1))
        list.append(node(newState, curnode.depth+1, 'R', curnode.depth+1 +
heuristic(newState, goal)))
    if x > 0:  # up
        newState = switch(curnode.data, curnode.blank, (x-1, y))
        list.append(node(newState, curnode.depth+1, 'U', curnode.depth+1 +
heuristic(newState, goal)))
    if y > 0:  # left
        newState = switch(curnode.data, curnode.blank, (x, y-1))
        list.append(node(newState, curnode.depth+1, 'L', curnode.depth+1 +
heuristic(newState, goal)))
    for child in list:
        child.prev = curnode
    return list

'''
Modeled after best-first search
'''
```

```
def a_star_search(rootNode, goal, heuristic):
    num_of_nodes = 0
    solution = []
    lookup = {}  # The dictionary will hash string representation of matrix
as key and node as value
    lookup[rootNode.string] = rootNode
    frontier = []  # list as frontier queue
    frontier.append(rootNode)
    num_of_nodes += 1
    while not len(frontier) == 0:
        cur = frontier.pop(0)
        if cur.data == goal:
            pointer = cur
            while pointer.prev != None:
                solution.append(pointer)
                pointer = pointer.prev  # backtracking
            return solution, num_of_nodes
        for child in expand(cur, heuristic, goal):
            s = child.string
            if s not in lookup or child.f < lookup[s].f:  # check for
repeated state
                lookup[s] = child
                search_add(frontier, child)
                num_of_nodes += 1
    print("Search failed")
    return solution, num_of_nodes

'''
Iterating through the frontier and insert the new node into the queue
'''
def search_add(list, node):
    done = False
    for i in range(len(list)):
        if node.f < list[i].f:
            list.insert(i,node)
            done = True
            break
    if not done:
        list.append(node)


if __name__ == '__main__':

    filename = input("Enter file name: ")
    file = open(filename,'r')
    lines = file.readlines()
    file.close()
    initial = [[0,0,0],[0,0,0],[0,0,0]]
    goal = [[0,0,0],[0,0,0],[0,0,0]]
    for i in range(3):
        nums = lines[i].split(' ')
        for j in range(3):
            initial[i][j] = nums[j][0]
    for n in range(4,7):
        nums = lines[n].split(' ')
        for m in range(3):
            goal[n-4][m] = nums[m][0]
```

```
    # As for which heuristic function to use
    heuristicModeStr = input("Which heuristic function? enter h1 or h2: ")
    if heuristicModeStr != 'h1' and heuristicModeStr != 'h2':
        raise ValueError("Unacceptable input.")
    if heuristicModeStr == 'h1':
        heuristicMode = h1
    else:
        heuristicMode = h2
    # Initiate root node
    rootNode = node(initial,0,None,heuristicMode(initial,goal))
    # Initiate search
    result, num_of_nodes = a_star_search(rootNode, goal, heuristicMode)

    # # For debug use
    # print(initial[0],initial[1],initial[2],sep='\n')
    # print()
    # print(goal[0],goal[1],goal[2], sep='\n')
    #
    # print(h1(initial,goal))
    # print(h2(initial,goal))
    #
    # print(rootNode.data[0],rootNode.data[1],rootNode.data[2], sep='\n')
    # print(str(rootNode.f) + ' ' + str(rootNode.depth))
    # for i in range(len(result)-1,-1,-1):
    #     nodes = result[i]
    #     print(nodes.data[0],nodes.data[1], nodes.data[2], sep='\n')
    #     print(nodes.move + ' ' + str(nodes.f) + ' ' + str(nodes.depth))

    outName = input("Enter output file name: ")
    end = open(outName + heuristicModeStr + '.txt', 'w')
    end.writelines(lines)
    end.write('\n')
    end.write(str(result[0].depth) + '\n')
    end.write(str(num_of_nodes) + '\n')
    # The result is in reverse order because the solution is traced from the
goal node back
    for i in range(len(result) - 1, -1, -1):
        end.write(result[i].move + ' ')
    end.write('\n')
    end.write(str(rootNode.f) + ' ')
    for i in range(len(result) - 1, -1, -1):
        end.write(str(result[i].f) + ' ')
    end.close()
```

Outputs:

Output1h1:

4 1 6

8 3 5

2 0 7

8 4 6

0 1 5

2 3 7

4

10

U U L D

4 4 4 4 4

Ouput1h2:

4 1 6

8 3 5

2 0 7

8 4 6

0 1 5

2 3 7

4

10

U U L D

31 25 16 16 4

Output2h1:

2 6 0

1 3 7

4 5 8

1 2 0

7 5 3

4 8 6

12

54

L D R U L L D R D R U U

10 10 12 12 12 12 12 12 12 12 12 12 12

Output2h2:

2 6 0

1 3 7

4 5 8

1 2 0

7 5 3

4 8 6

12

42

L D R U L L D R D R U U

49 43 51 45 45 33 45 39 33 24 24 24 12

Output3h1:

8 6 3

0 4 5

7 2 1

1 2 3

4 0 7

6 5 8

25

2542

U R D D R U L D L U U R D R D L L U U R D R D L U

19 19 19 21 21 21 21 23 23 23 23 23 23 23 23 25 25 25 25 25 25 25 25 25 25 25

Output3h2:

8 6 3

0 4 5

7 2 1

1 2 3

4 0 7

6 5 8

25

97

U R D R D L L U R R D L L U U R D R D L L U U R D

58 52 52 51 48 54 50 56 50 43 46 46 52 52 52 52 46 43 40 40 40 40 40 40 34 25