Python 3.7.4

How To Run:

1. Create folder with python file (GraphSearchAClean.py) and input file as a txt inside
2. Open python file and rename the string assigned to variable txt in line 14 to the name of the input file.
3. Run the python file.
4. An output file will be created in the folder and the python shell will display the output as well.

Code:

```
import copy

'''
Daniel Tse
Professor E.K. Wong

CS 4613

GRAPH-SEARCH Algorithm with A* Search Strategy
Solving: 14-puzzle problem
'''
#assign txt to file name of input file
txt = "Input3.txt"
file = open("Output" + txt[5] + ".txt", "w")

#lists are used to represent inititial and goal states
#indicies represent tile numbers
#values associated with keys are lists containing tile position
target = []
array = []
generated = []
side_size = 4
empty = 0       #value of empty

num_generated = 0

#node for tree
class node:
    def __init__(self, state = None, depth = 0, action = None, prev_node = None):
        self.state = state          #state the node represents
        self.depth = depth          #g(n)
        self.est = est_goal(state)  #h(n)
        self.action = action        #action taken to arrive at node
        self.prev_node = prev_node  #previous node

#calculate estimate to reach goal
def est_goal(start):
    total = 0

    #get manhattan distance of all other tiles
    for i in range (1, 15):
        #print(i, end = ': ')
        total += manhattan(start[i][0], target[i][0])
    return total

#calculate manhattan distance of two points
def manhattan(start_pos, target_pos):
    pos_diff = abs(start_pos - target_pos)
    y_diff = abs(start_pos // side_size - target_pos // side_size)
    x_diff = abs((min(start_pos, target_pos) + side_size * y_diff) -  max(start_pos, target_pos))
    total = (x_diff + y_diff)
    #print(total)
    return total

#returns the tile value at the given position at the given state
def tileAtPos(curr_state, pos):
    for i in range(len(curr_state)):
        for j in curr_state[i]:
            if j == pos:
                return i
```

```
#input 0: output: 1; input 1: output 0
#can only take 1 or 0 as input
def opposite(num):
    if num == 0:
        return 1
    return 0

#return new state if value top of empty spot is moved into empty spot
def upAction(state, num):
    curr_state = copy.deepcopy(state)
    #variable pos is the position of the potential tile to be swapped
    pos = curr_state[empty][num] - 4
    #check if potential position is out of bounds
    if (pos) < 0:
        return None
    tile = tileAtPos(state, pos)
    curr_state[empty][num], curr_state[tile][0] = curr_state[tile][0], curr_state[empty][num]
    return curr_state

#return new state if value right of empty spot is moved into empty spot
def rightAction(state, num):
    curr_state = copy.deepcopy(state)
    #variable pos is the position of the potential tile to be swapped
    pos = curr_state[empty][num] + 1
    #check if potential position is out of bounds
    if (curr_state[empty][num] - 3) % 4 == 0:
        return None
    tile = tileAtPos(curr_state, pos)
    curr_state[empty][num], curr_state[tile][0] = curr_state[tile][0], curr_state[empty][num]
    return curr_state

#return new state if value down of empty spot is moved into empty spot
def downAction(state, num):
    curr_state = copy.deepcopy(state)
    #variable pos is the position of the potential tile to be swapped
    pos = curr_state[empty][num] + 4
    #check if potential position is out of bounds
    if (pos) > 15:
        return None
    tile = tileAtPos(curr_state, pos)
    curr_state[empty][num], curr_state[tile][0] = curr_state[tile][0], curr_state[empty][num]
    return curr_state

#return new state if value right of empty spot is moved into empty spot
def leftAction(state, num):
    curr_state = copy.deepcopy(state)
    #variable pos is the position of the potential tile to be swapped
    pos = curr_state[empty][num] - 1
    #check if potential position is out of bounds
    if (curr_state[empty][num]) % 4 == 0:
        return None
    tile = tileAtPos(curr_state, pos)
    curr_state[empty][num], curr_state[tile][0] = curr_state[tile][0], curr_state[empty][num]
    return curr_state
```

```python
#displays puzzle given current state
def display(curr_state):
    if curr_state == None:
        print("invalid")
        return
    #populate list with values
    arr = [None] * 16
    for i in range(15):
        for pos in curr_state[i]:
            arr[pos] = i
    j = 0
    for j in range(16):
        print('{:>3d}'.format(arr[j]), end =" ")
        file.write(str(arr[j]) + " ")
        if (j + 1) % 4 == 0:
            print("\n")
            file.write("\n")
    print("\n")

def displayfrontier(frontier):
    for node in frontier:
        display(node.state)

#returns node with lowest f(n)
def lowestf(frontier):
    min_node = frontier[0]
    for node in frontier:
        if (node.depth + node.est) < (min_node.depth + min_node.est):
            min_node = node
    return min_node

#returns new frontier after expanding node and checking for repeated states
def expand(node, frontier):

    #potential states from first empty spot
    frontier = expand_helper(node, frontier, 0)
    #potential states from second empty spot
    frontier = expand_helper(node, frontier, 1)
    return frontier

def expand_helper(OGnode, frontier, space):
    up = upAction(OGnode.state, space)
    right = rightAction(OGnode.state, space)
    down = downAction(OGnode.state, space)
    left = leftAction(OGnode.state, space)

    if up != None:
        curr = node(up, OGnode.depth + 1, ("U" + str(space + 1)), OGnode)
        frontier = check_repeat(curr, frontier)
    if right != None and not (stateEqual(target, frontier[-1].state)):
        curr = node(right, OGnode.depth + 1, ("R" + str(space + 1)), OGnode)
        frontier = check_repeat(curr, frontier)
    if down != None and not (stateEqual(target, frontier[-1].state)):
        curr = node(down, OGnode.depth + 1, ("D" + str(space + 1)), OGnode)
        frontier = check_repeat(curr, frontier)
    if left != None and not (stateEqual(target, frontier[-1].state)):
        curr = node(left, OGnode.depth + 1, ("L" + str(space + 1)), OGnode)
        frontier = check_repeat(curr, frontier)
    return frontier
```

```
def check_repeat(node, frontier):
    if stateEqual(node.state, target):
        frontier.append(node)
        generated.append(node.state)
        return frontier
    #a cheaper path to the state has already been found - no action taken
    for curr in generated:
        if stateEqual(node.state, curr):
            return frontier
    frontier.append(node)
    generated.append(node.state)
    return frontier

#recursive A* Search - generates tree and returns goal node
def search(frontier):
    for node in frontier:
        if stateEqual(node.state, target):
        #if node.state == target:
            return node
    curr = lowestf(frontier)
    frontier = expand(curr, frontier)
    frontier.remove(curr)
    return(search(frontier))

#returns true if two states are equal, false otherwise
def stateEqual(state1, state2):
    return state1[1:] == state2[1:]

#returns tuple of lists - one for action sequence and one for f(n) values of solution path
#out is a list containing two lists: first list contains f(n) values, second list contains actions
def getPath(node, out):
    out[0].append(node.depth + node.est)
    if (node.prev_node == None):
        return out
    out[1].append(node.action)
    return(getPath(node.prev_node, out))
```

```python
def main():
    #open file and read all values into array
    f = open(txt, 'r')
    array = ([int(x) for x in f.read().split()])

    #update list with positions ranging 0-15 according to tile
    initial = []

    for i in range(15):
        initial.append([])
        target.append([])

    for i in range(16):
        initial[array[i]].append(i)
        target[array[i + 16]].append(i)

    #assign root node and populate frontier, and generated node list
    root = node(initial)
    actions = []
    frontier = [root]
    generated.append(root.state)

    #call search function and return final node to fin
    fin = search(frontier)

    #display output
    print("Initial:")
    display(initial)
    file.write("\n")
    print("Target:")
    display(target)
    file.write("\n")


    print("Depth: ", fin.depth, "\n")
    file.write(str(fin.depth) + "\n")
    print("Number of generated nodes: ", len(generated), "\n")
    file.write(str(len(generated)) + "\n")

    emptyOutput = [[],[]]
    output = getPath(fin, emptyOutput)

    #display actions
    print("actions: ")
    for f in reversed(output[1]):
        print(f, end =" ")
        file.write(f + " ")
    print("\n")
    file.write("\n")

    #display f values
    print("f values: ")
    for action in reversed(output[0]):
        print(action, end =" ")
        file.write(str(action) + " ")
    print("\n")

    file.close()
main()
```

Output 1

```
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
61
L1 D1 D1 U2 U2 R2
6 6 6 6 6 6 6
```

Output 2

```
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
436
R2 R1 R1 D1 D1 L1 U2 U2 R2 D2 D2 L2
10 10 12 12 12 12 12 12 12 12 12 12 12
```

Output 3

```
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
300
U1 L1 D1 L1 D1 D2 R1 U1 R1 R1 R2 R2 U2 L2
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
```