Michael Kwan
Artificial Intelligence
Project 1
Instructions on How to Run Program

You can compile the program using an IDE with the source code in a python file. You should also have the input files in the same folder as the python file. You can run it as a python program and it will ask you to input the name of the input file and the name of the output file that you and will respectively read from and write to. After you run the program, the output file will be in the same folder as the python file.

I have copy and pasted the source code, sample_output, output1, output2, and output3 below.

Source Code:

```
import heapq, copy

def readInput(filename, initial, goal):
   inputFile = open(filename, "r")

   #read first three lines as initial state
   for i in range(3):
      initial.append(inputFile.readline().strip().split(" "))

   #skip the fourth line, which is blank
   inputFile.readline()

   #read last three lines as goal state
   for i in range(3):
      goal.append(inputFile.readline().strip().split(" "))

   inputFile.close()

def writeOutput(filename, initial, goal, d, N, actions, fVals):
   outputFile = open(filename, "w")

   #write initial state
   for i in range(3):
      outputFile.write(" ".join(initial[i]))
      outputFile.write("\n")
   outputFile.write("\n")

   #write goal state
   for i in range(3):
      outputFile.write(" ".join(goal[i]))
```

```
        outputFile.write("\n")
    outputFile.write("\n")

    #write d and N
    outputFile.write(str(d))
    outputFile.write("\n")
    outputFile.write(str(N))
    outputFile.write("\n")

    #write actions
    for action in actions:
        outputFile.write(str(action))
        outputFile.write(" ")
    outputFile.write("\n")

    #write fVals
    for fVal in fVals:
        outputFile.write(str(fVal))
        outputFile.write(" ")
    outputFile.write("\n")

    outputFile.close()

def calcManhattanDist(currState, goalState):
    #find compare each cell in currState with goalState to find and sum all manhattan distances
    dist = 0
    for cRow in range(len(currState)):
        for cCol in range(len(currState[0])):
            for gRow in range(len(goalState)):
                for gCol in range(len(goalState[0])):
                    if currState[cRow][cCol] == goalState[gRow][gCol]:
                        dist = dist + abs(gRow-cRow)+abs(gCol-cCol)
    return dist

def stateAfterAction(state, action, blankRow, blankCol):
    #Depending on which action is being taken, move the blank cell and return the modified state
    if action == "U":
        state[blankRow][blankCol],state[blankRow-1][blankCol] =
state[blankRow-1][blankCol],state[blankRow][blankCol]
    elif action == "D":
        state[blankRow][blankCol],state[blankRow+1][blankCol] =
state[blankRow+1][blankCol],state[blankRow][blankCol]
    elif action == "L":
        state[blankRow][blankCol],state[blankRow][blankCol-1] =
state[blankRow][blankCol-1],state[blankRow][blankCol]
    elif action == "R":
        state[blankRow][blankCol],state[blankRow][blankCol+1] =
state[blankRow][blankCol+1],state[blankRow][blankCol]
    else:
        print("Invalid Action")
    return state

def solver(prioQueue, seenStates, N):
    while len(prioQueue) > 0:
        #pop the first node in the priority queue sorted by its f value and get all of its information
        node = heapq.heappop(prioQueue)
        fVal = node[0]
        state = node[1]
        goal = node[2]
        blankRow = node[3]
        blankCol = node[4]
        actions = node[5]
        fVals = node[6]

        if state == goal:
            #the goal has been reached so we return all the actions taken, the f values of all the
nodes, and the total nodes visited
            return (actions, fVals, N)
        else:
            if blankRow != 0:
                #if the blank space can move up, we move it by making a copy of all the relevant
information to be modified
                stateCopy = copy.deepcopy(state)
                newState = stateAfterAction(stateCopy, "U", blankRow, blankCol)
                strState = str(newState)
                if strState not in seenStates:
                    #check to make sure state has not already been visited already because this is a
graph search
                    #if the state has not been visited, then make new modifications and add new mode
to priority queue
                    seenStates[strState] = True
                    hVal = calcManhattanDist(newState,goal)
                    newActions = copy.copy(actions)
                    newActions.append("U")
                    gVal = len(newActions)
                    fVal = gVal + hVal
                    newfVals = copy.copy(fVals)
                    newfVals.append(fVal)
                    newElement = [fVal, newState, goal, blankRow-1, blankCol, newActions, newfVals]
                    heapq.heappush(prioQueue, newElement)
                    N+=1
            if blankRow != (len(state)-1):
                #if the blank space can move down, we move it by making a copy of all the relevant
information to be modified
                stateCopy = copy.deepcopy(state)
                newState = stateAfterAction(stateCopy, "D", blankRow, blankCol)
                strState = str(newState)
                if strState not in seenStates:
                    #check to make sure state has not already been visited already because this is a
graph search
                    #if the state has not been visited, then make new modifications and add new mode
to priority queue
                    seenStates[strState] = True
                    hVal = calcManhattanDist(newState,goal)
                    newActions = copy.copy(actions)
                    newActions.append("D")
                    gVal = len(newActions)
                    fVal = gVal + hVal
                    newfVals = copy.copy(fVals)
                    newfVals.append(fVal)
                    newElement = [fVal, newState, goal, blankRow+1, blankCol, newActions, newfVals]
                    heapq.heappush(prioQueue, newElement)
                    N+=1
            if blankCol != 0:
                #if the blank space can move left, we move it by making a copy of all the relevant
information to be modified
                stateCopy = copy.deepcopy(state)
                newState = stateAfterAction(stateCopy, "L", blankRow, blankCol)
                strState = str(newState)
                if strState not in seenStates:
                    #check to make sure state has not already been visited already because this is a
graph search
                    #if the state has not been visited, then make new modifications and add new mode
to priority queue
                    seenStates[strState] = True
                    hVal = calcManhattanDist(newState,goal)
                    newActions = copy.copy(actions)
                    newActions.append("L")
                    gVal = len(newActions)
                    fVal = gVal + hVal
                    newfVals = copy.copy(fVals)
                    newfVals.append(fVal)
                    newElement = [fVal, newState, goal, blankRow, blankCol-1, newActions, newfVals]
                    heapq.heappush(prioQueue, newElement)
                    N+=1
            if blankCol != (len(state[0])-1):
                #if the blank space can move right, we move it by making a copy of all the relevant
information to be modified
                stateCopy = copy.deepcopy(state)
                newState = stateAfterAction(stateCopy, "R", blankRow, blankCol)
                strState = str(newState)
                if strState not in seenStates:
                    #check to make sure state has not already been visited already because this is a
graph search
                    #if the state has not been visited, then make new modifications and add new mode
to priority queue
                    seenStates[strState] = True
                    hVal = calcManhattanDist(newState,goal)
                    newActions = copy.copy(actions)
                    newActions.append("R")
                    gVal = len(newActions)
                    fVal = gVal + hVal
                    newfVals = copy.copy(fVals)
                    newfVals.append(fVal)
                    newElement = [fVal, newState, goal, blankRow, blankCol+1, newActions, newfVals]
                    heapq.heappush(prioQueue, newElement)
                    N+=1
    return False

def main():
    #get file names to be used as input and output
    inputFileName = input("Please enter the input filename: ")
    outputFileName = input("Please enter the output filename: ")
    #initialize variables to create the first node for the initial state
    initial = []
    goal = []
    seenStates = {}
    H = []
    heapq.heapify(H)
    N = 1
    actions = []
    readInput(inputFileName, initial, goal)
    state = copy.deepcopy(initial)
    initialVal = calcManhattanDist(state,goal)
    fVals = [initialVal]
    seenStates[str(state)] = True
    #find blankRow and blankCol of initial state
    for i in range(len(state)):
        for j in range(len(state[0])):
            if state[i][j] == "0":
                blankRow = i
                blankCol = j

    #push the first node onto the priotity queue
    firstElement = [initialVal, state, goal, blankRow, blankCol, actions, fVals]
    heapq.heappush(H, firstElement)

    #call the solver to find the solution, which returns the actions taken, f values, and total nodes created
    res = solver(H, seenStates, N)

    if res == False:
        #the res will be false if the priority queue has no nodes left, which means it has explored everything possible from the initial state
        print("Goal could not be reached")
    else:
        #extract all the info from the res and output it into thte designated output file
        actions = res[0]
        fVals = res[1]
        newN = res[2]
        d = len(actions)
        writeOutput(outputFileName, initial, goal, d, newN, actions, fVals)

#call main to start the program
main()
```

Sample_Output:

```
5 6 0 7
8 9 10 11
2 3 4 1

5 9 6 7
8 3 0 11
2 4 10 1

5
N
L D D R U
f f f f f
```

Output1:

```
8 0 6 4
3 10 7 9
11 5 2 1

8 10 6 4
0 3 2 9
11 7 5 1

6
22
D R D L U L
8 7 8 9 8 7 6
```

Output2:

```
8 0 6 4
3 10 7 9
11 5 2 1

8 7 2 4
10 6 9 1
3 11 5 0

12
36
R D D L L U R U R D R D
16 15 14 13 14 15 16 15 16 15 14 13 12
```

Output3:

```
8 7 2 4
10 6 9 1
0 11 5 3

10 6 8 4
9 0 7 2
11 5 3 1

16
107
R U R U L L D R D R R U L U L D
16 15 16 17 18 17 18 17 16 17 18 19 18 17 18 17 16
```