JunHao Chen
Artificial Intelligence Project - 14 Puzzle problem
November 5, 2019

Instruction for Running Program

The driver() function takes in an input file to generate an output file that contains the solution for the goal node. The driver() function will automatically generate an output file that contains the same name as the input file. For example, input1.txt will generate output1.txt. The program will automatically assume that input1.txt, input2.txt and input3.txt already exists in the file directory and will output solutions for each respectively. To test new input files, simply add a new line that calls driver() with the input file. For example, to test input4.txt, add driver("input4.txt").

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
74
L1 , D1 , D1 , U2 , U2 , R2
6 , 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
584
R2 , R1 , R1 , D1 , U2 , U2 , R2 , D2 , D2 , L2 , D1 , L1
10 , 10 , 12 , 12 , 12 , 12 , 12 , 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
400
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
```

Source Code

```
import copy

#sets up by parsing the file and converting the initial and goal board states to 2D arrays
#convert 0 to unique blanks (B1,B2)
def setup(filename):
  lines = []
  with open(filename) as file:
     for line in file:
        line = line.strip()
        line = line.split()
        lines.append(line)
  goalState = lines[5:]
  initialState = lines[:4]
  count = 1
  for i in range(len(initialState)):
     for j in range(len(initialState[i])):
        if initialState[i][j] == '0':
           initialState[i][j] = "B"+str(count)
           count+=1
  return goalState, initialState

# node class
# parent = parent node
# depth = depth of tree = g(n) = total path cost
# moves = moves to get to the board state (L,R,R)
# hnVal = heuristic = manhattan distance computed
# fnVal = g(n) + h(n)
class Node:
  def __init__(self, state, parent, move , hnVal):
     self.state = state
     self.parent = parent
     self.hnVal = hnVal
     self.zeroState = deUnique(self.state)

     if parent is None:
        self.depth = 0
        self.moves = move
        else:
            self.depth = parent.depth + 1
            if parent.moves == '':
                self.moves = move
            else:
                self.moves = parent.moves + " , " + move

        self.fnVal = self.hnVal + self.depth

        if parent is not None:
            self.fnList = parent.fnList + " , " + str(self.fnVal)
        else:
            self.fnList = str(self.fnVal)

        #to sort in priority queue based on fnVal
    def __lt__(self, other):
        return self.fnVal < other.fnVal

   #each node is represented by board state
    def __repr__(self):
        return str(self.state)

    def __str__(self):
        return str(self.state)

    def actions(self):
        #state of each action
        actionStates = []
        zeroCoord = []
        #action performed
        actions= []
        inst = []

        #convert the zeros into coordinates
        for i in range(len(self.state)):
            for j in range(len(self.state[i])):
                if self.state[i][j] == 'B1' or self.state[i][j]== 'B2':
                    zeroCoord.append((i,j))
```

```
#perform possible SINGLE (U,D,L,R) for each blank space
for i in zeroCoord:
    x = i[0]
    y = i[1]
    if self.state[x][y] == 'B1':
        blankNum = '1'
    else:
        blankNum = '2'

    # LEFT, check if pos y = 0 for invalid boundary
    if y != 0:
        #makes a copy so each move does not influence each other
        temp = copy.deepcopy(self.state)
        #move left by swapping
        temp[x][y-1],temp[x][y] = temp[x][y],temp[x][y-1]
        actionStates.append(temp)
        inst.append("L"+blankNum)


    # RIGHT, check if pos y = 3 for invalid boundary
    if y != 3:
        #makes a copy so each move does not influence each other
        temp = copy.deepcopy(self.state)
        #move left by swapping
        temp[x][y+1],temp[x][y] = temp[x][y],temp[x][y+1]
        actionStates.append(temp)
        inst.append("R"+blankNum)


    # UP, check if pos x = 0 for invalid boundary
    if x != 0:
        #makes a copy so each move does not influence each other
        temp = copy.deepcopy(self.state)
        #move left by swapping
        temp[x-1][y],temp[x][y] = temp[x][y],temp[x-1][y]
        actionStates.append(temp)
        inst.append("U"+blankNum)
    # DOWN, check if pos x = 3 for invalid boundary
    if x != 3:
        #makes a copy so each move does not influence each other
        temp = copy.deepcopy(self.state)
        #move left by swapping
        temp[x+1][y],temp[x][y] = temp[x][y],temp[x+1][y]
        actionStates.append(temp)
        inst.append("D"+blankNum)


#perform combination move if zeroes are next to each other
x1 = zeroCoord[0][0]
y1 = zeroCoord[0][1]
x2 = zeroCoord[1][0]
y2 = zeroCoord[1][1]
if self.state[x1][y1] == 'B1':
    num1 = '1'
    num2 = '2'

else:
    num1 = '2'
    num2 = '1'

#check LEFT DOUBLE MOVE. abs(y1 - y2) checks if they're next to each other
if abs(y1 - y2) == 1 and x1 == x2:
    # checks if y2 is on the left of y1 or vice versa
    # check if pos y = 0 for invalid boundary
    if y2 < y1 and y2 != 0:
        temp = copy.deepcopy(self.state)
        #move left most blank tile
        temp[x2][y2 - 1], temp[x2][y2] = temp[x2][y2], temp[x2][y2-1]
        #move the other tile
        temp[x2][y1 - 1], temp[x2][y1] = temp[x2][y1], temp[x2][y1-1]
        inst.append('L' + num2 + 'L' + num1)
        actionStates.append(temp)
    elif y1 < y2 and y1 != 0 :
        temp = copy.deepcopy(self.state)
        #move left most blank tile
        temp[x2][y1 - 1], temp[x2][y1] = temp[x2][y1], temp[x2][y1-1]
```

```
            #move the other tile
            temp[x2][y2 - 1], temp[x2][y2] = temp[x2][y2], temp[x2][y2-1]
            inst.append('L' + num1 + 'L' + num2)
            actionStates.append(temp)

    #check RIGHT DOUBLE MOVE. abs(y1 - y2) checks if they're next to each other
    if abs(y1 - y2) == 1 and x1 == x2:
        # checks if y2 is on the RIGHT of y1 or vice versa
        # check if pos y = 3 for invalid boundary
        if y2 > y1 and y2 != 3:
            temp = copy.deepcopy(self.state)
            #move RIGHT most blank tile
            temp[x2][y2 + 1], temp[x2][y2] = temp[x2][y2], temp[x2][y2+1]
            #move the other tile
            temp[x2][y1 + 1], temp[x2][y1] = temp[x2][y1], temp[x2][y1+1]
            inst.append('R' + num2 + 'R' + num1)
            actionStates.append(temp)

        elif y1 > y2 and y1 != 3 :
            temp = copy.deepcopy(self.state)
            #move RIGHT most blank tile
            temp[x2][y1 + 1], temp[x2][y1] = temp[x2][y1], temp[x2][y1+1]
            #move the other tile
            temp[x2][y2 + 1], temp[x2][y2] = temp[x2][y2], temp[x2][y2+1]
            inst.append('R' + num1 + 'R' + num2)
            actionStates.append(temp)


    #check UP DOUBLE MOVE. abs(x1 - x2) checks if they're next to each other
    if abs(x1 - x2) == 1 and y1 == y2:
        # checks if x2 is on the TOP of x1 or vice versa
        # check if pos x = 0 for invalid boundary
        if x2 < x1 and x2 != 0:
            temp = copy.deepcopy(self.state)
            #move TOP most blank tile
            temp[x2 - 1][y2], temp[x2][y2] = temp[x2][y2], temp[x2 - 1][y2]
            #move the other tile
            temp[x1 - 1][y1], temp[x1][y1] = temp[x1][y1], temp[x1 - 1][y1]
            inst.append('U' + num2 + 'U' + num1)
            actionStates.append(temp)

        elif x1 < x2 and x1 != 0 :
            temp = copy.deepcopy(self.state)
            #move TOP most blank tile
            temp[x1 - 1][y1], temp[x1][y1] = temp[x1][y1], temp[x1 - 1][y1]
            #move the other blank tile
            temp[x2 - 1][y2], temp[x2][y2] = temp[x2][y2], temp[x2 -1][y2]
            inst.append('U' + num1 + 'U' + num2)
            actionStates.append(temp)



    #check DOWN DOUBLE MOVE. abs(x1 - x2) checks if they're next to each other
    if abs(x1 - x2) == 1:
        # checks if x2 is on the TOP of x1 or vice versa
        # check if pos x = 3 for invalid boundary
        if x2 > x1 and x2 != 3 and y1 == y2:
            temp = copy.deepcopy(self.state)
            #move BOTTOM most blank tile
            temp[x2 + 1][y2], temp[x2][y2] = temp[x2][y2], temp[x2 + 1][y2]
            #move the other tile
            temp[x1 + 1][y1], temp[x1][y1] = temp[x1][y1], temp[x1 + 1][y1]
            inst.append('D' + num2 + 'D' + num1)
            actionStates.append(temp)

        elif x1 > x2 and x1 != 3 :
            temp = copy.deepcopy(self.state)
            #move BOTTOM most blank tile
            temp[x1 + 1][y1], temp[x1][y1] = temp[x1][y1], temp[x1 + 1][y1]
            #move the other blank tile
            temp[x2 + 1][y2], temp[x2][y2] = temp[x2][y2], temp[x2 + 1][y2]
            inst.append('D' + num1 + 'D' + num2)
            actionStates.append(temp)

    return actionStates,inst

class Queue:
  def __init__(self):
        self.items=[]

    def __repr__(self):
        return str(self.items)

    def __str__(selfs):
        return str(self.items)

    def __len__(self):
        return len(self.items)

    def pop(self):
        return self.items.pop(0)

# prioirity queue by append item and then sorting based on fnVal
    def append(self,item):
        self.items.append(item)
        self.items.sort()

# translate 2D matrix into coordinates to so computing manhattan dist is easier
def coordinate(state):
    coords = {}
    for i in range(len(state)):
        for j in range(len(state[i])):
            coords[ state[i][j] ] = (i,j)
    return coords

# convers unique blanks (B1,B2) to just 0 for checking repeated board states
def deUnique(state):
     temp = copy.deepcopy(state)
     for i in range(len(temp)):
         for j in range(len(temp[i])):
             if temp[i][j] == "B1" or temp[i][j] == "B2":
                 temp[i][j] = "0"
     return temp


#computes manhattan dist of currentState and goalState
def manhattanDist(currentState,goalState):
    state = coordinate(currentState)
    goal = coordinate(goalState)
    result=0
    for i in range(len(currentState)):
        for j in range(len(currentState)):
            if (currentState[i][j] != '0'):
                stateCoord = state[currentState[i][j]]
                goalCoord = goal[currentState[i][j]]
                result += abs(stateCoord[0] - goalCoord[0]) + abs(stateCoord[1] -
goalCoord[1])
    return result

# goal test to see if 2D arr in currentState is the same as goalState
def goalTest(currentState,goalState):
    return currentState == goalState

#translate board state to tuples so it can be stored in dict. Faster access for dups
def tuplify(state):
    temp = copy.copy(state)
    for i in range(len(temp)):
        temp[i] = tuple(temp[i])
    temp = tuple(temp)
    return temp

#return -1 if node not in frontier. Else returns the index
def frontierSearch(frontier,node):
    for item in frontier:
        if item.zeroState == node.zeroState:
            return frontier.index(item)
    return -1




# takes in a startNode with the inital board state and a 2D array of the goalState
def A_Star_Search(startNode,goalState):
    frontier = Queue()
    explored = {}
    frontier.append(startNode)
```

```
  count = 0

  while len(frontier)!=0:
     node = frontier.pop()
     print(node)
     if goalTest(node.zeroState,goalState):
        return node,count
     # add the current board state to explored which is a dict
     state = tuplify(node.zeroState)
     explored[state] = state
     actionState, inst = node.actions()
     for action in actionState:
        # make nodes only if the state has not been explored
        if tuplify(deUnique(action)) not in explored:
           someNode =
Node(action,node,inst[actionState.index(action)],manhattanDist(deUnique(action),goalS
tate))
           count += 1
           index = frontierSearch(frontier.items,someNode)
           # found same state in frontier
           if index > 0 :
              # checks if node of same state in frontier's f(n) is greater than new node's
f(n)
              if frontier.items[index].fnVal > someNode.fnVal:
                 # replace node if it is
                 frontier.items.pop(index)
                 frontier.append(someNode)
           # state not found in frontier, add it
           else:
              frontier.append(someNode)

  return None,count

# used to output the result onto text file
def convertOutput(initialState,goalState,node,count,file):
  for i in deUnique(initialState):
     for j in i:
        print(j,end=" ",file=file)
     print("",file=file)
```

```
    print("",file=file)

    for i in goalState:
        for j in i:
            print(j,end=" ",file=file)
        print("",file=file)

    print("",file=file)
    print(str(node.depth),file=file)
    print(str(count),file=file)
    print(node.moves,file=file)
    print(node.fnList,file=file)
    file.close()

#driver program
def driver(inputFile):
    goalState,initialState = setup(inputFile)
    hnVal  = manhattanDist(deUnique(initialState),goalState)
    startNode = Node(initialState, None,'', hnVal)
    result,count = A_Star_Search(startNode,goalState)
    output = inputFile.split(".")
    # output file name = input file name count
    outputFile = "Output" + output[0][-1] + ".txt"
    file = open(outputFile, "w")
    convertOutput(initialState,goalState,result,count,file)


driver("Input1.txt")
driver("Input2.txt")
driver("Input3.txt")
```