**Name: Ashutosh Raj Gupta**

**Roll no: TECO2223A045**

**Batch: A3**

**ASSIGNMENT 2: Implement A star Algorithm for any game search problem**

**Code:1.8Puzzle**

class Node:

    def \_\_init\_\_(self, data, level, fval):

        # Initialize the node with the data ,level of the node and the calculated fvalue

        self.data = data

        self.level = level

        self.fval = fval

    def generate\_child(self):

        # Generate hild nodes from the given node by moving the blank space

        # either in the four direction {up,down,left,right}

        x, y = self.find(self.data, '\_')

        # val\_list contains position values for moving the blank space in either of

        # the 4 direction [up,down,left,right] respectively.

        val\_list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]

        children = []

        for i in val\_list:

            child = self.shuffle(self.data, x, y, i[0], i[1])

            if child is not None:

                child\_node = Node(child, self.level + 1, 0)

                children.append(child\_node)

        return children

    def shuffle(self, puz, x1, y1, x2, y2):

        # Move the blank space in the given direction and if the position value are out

        # of limits the return None

        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):

            temp\_puz = []

            temp\_puz = self.copy(puz)

            temp = temp\_puz[x2][y2]

            temp\_puz[x2][y2] = temp\_puz[x1][y1]

            temp\_puz[x1][y1] = temp

            return temp\_puz

        else:

            return None

    def copy(self, root):

        # copy function to create a similar matrix of the given node

        temp = []

        for i in root:

            t = []

            for j in i:

                t.append(j)

            temp.append(t)

        return temp

    def find(self, puz, x):

        # 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 puz[i][j] == x:

                    return i, j

class Puzzle:

    def \_\_init\_\_(self, size):

        # Initialize the puzzle size by the the specified size,open and closed lists to empty

        self.n = size

        self.open = []

        self.closed = []

    def accept(self):

        # Accepts the puzzle from the user

        puz = []

        for i in range(0, self.n):

            temp = input().split(" ")

            puz.append(temp)

        return puz

    def f(self, start, goal):

        # Heuristic function to calculate Heuristic value f(x) = h(x) + g(x)

        return self.h(start.data, goal) + start.level

    def h(self, start, goal):

        # Calculates the difference between the given puzzles

        temp = 0

        for i in range(0, self.n):

            for j in range(0, self.n):

                if start[i][j] != goal[i][j] and start[i][j] != '\_':

                    temp += 1

        return temp

    def process(self):

        # Accept Start and Goal Puzzle state

        print("enter the start state matrix \n")

        start = self.accept()

        print("enter the goal state matrix \n")

        goal = self.accept()

        start = Node(start, 0, 0)

        start.fval = self.f(start, goal)

        # put the start node in the open list

        self.open.append(start)

        print("\n\n")

        while True:

            cur = self.open[0]

            print("==================================================\n")

            for i in cur.data:

                for j in i:

                    print(j, end=" ")

                print("")

            # if the difference between current and goal node is 0 we have reached the goal node

            if (self.h(cur.data, goal) == 0):

                break

            for i in cur.generate\_child():

                i.fval = self.f(i, goal)

                self.open.append(i)

            self.closed.append(cur)

            del self.open[0]

            # sort the open list based on f value

            self.open.sort(key=lambda x: x.fval, reverse=False)

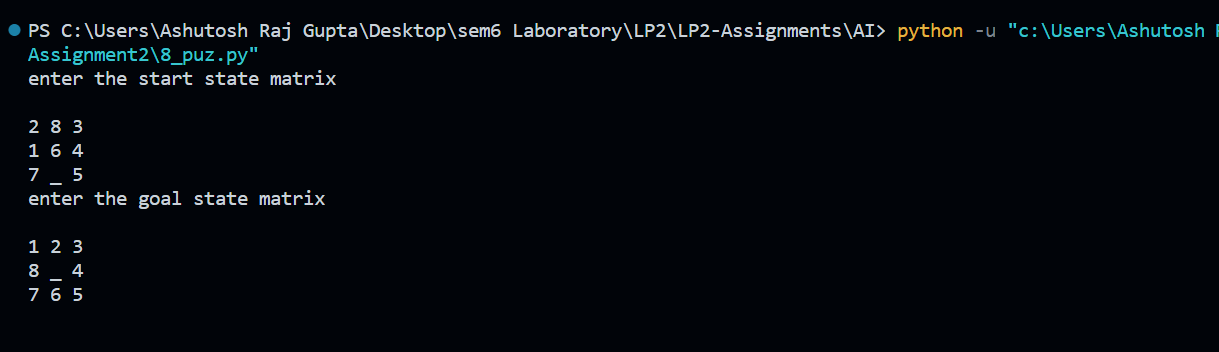
# matrix 3X3

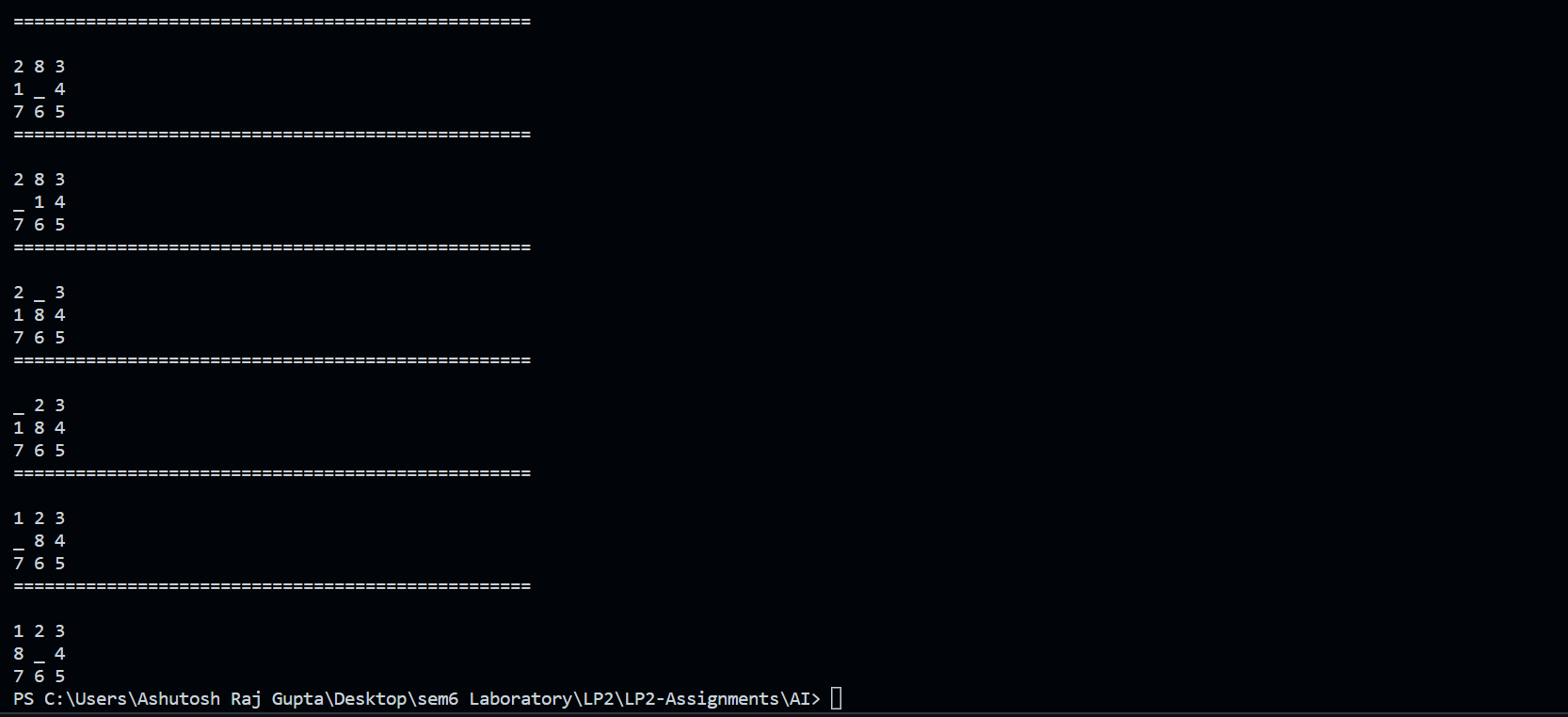
puz = Puzzle(3)

puz.process()

#for each blank space move 3 child will be calculated and from that 3 puzzle best f value will be selected

**Output:**

****

****

**2. A star Algorithm**

def aStarAlgo(start\_node, stop\_node):

    open\_set = set(start\_node)

    closed\_set = set()

    g = {}               #store distance from starting node

    parents = {}         # parents contains an adjacency map of all nodes

    #distance of starting node from itself is zero

    g[start\_node] = 0

    #start\_node is root node i.e it has no parent nodes

    parents[start\_node] = start\_node

    while len(open\_set) > 0:

        n = None

        #node with lowest f() is found

        for v in open\_set:

            #if first node then n==none otherwise for other node we are calculating f value and comparing current node and previuos node f value

            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

                n = v

        if n == stop\_node or Graph\_nodes[n] == None:

            pass

        #for intermediate node

        else:

            for (m, weight) in get\_neighbors(n):

                # if we are encountering new node

                if m not in open\_set and m not in closed\_set:

                    open\_set.add(m)

                    parents[m] = n

                    g[m] = g[n] + weight

                #for each node m,compare its distance from start i.e g(m) to the

                #from start through n node

                else:

                    # if new node path is greater than previous then we are updating path with least cost

                    if g[m] > g[n] + weight:

                        g[m] = g[n] + weight

                        #change parent of m to n

                        parents[m] = n

                        #if m in closed set,remove and add to open

                        if m in closed\_set:

                            # need to explore again that why added in open\_list

                            closed\_set.remove(m)

                            open\_set.add(m)

        if n == None:

            print('Path does not exist!')

            return None

        # if the current node is the stop\_node then we can go back to the path from it to the start

        if n == stop\_node:

            path = []

            while parents[n] != n:

                path.append(n)

                n = parents[n]

            path.append(start\_node)

            path.reverse()

            print('Path found: {}'.format(path))

            return path

        # remove n from the open\_list, and add it to closed\_list

        # because all of his neighbors were inspected

        open\_set.remove(n)

        closed\_set.add(n)

    print('Path does not exist!')

    return None

def get\_neighbors(v):

    if v in Graph\_nodes:

        return Graph\_nodes[v]

    else:

        return None

def heuristic(n):

    H\_dist = {

        'A': 11,

        'B': 6,

        'C': 99,

        'D': 1,

        'E': 7,

        'G': 0,

    }

    return H\_dist[n]

Graph\_nodes = {

    'A': [('B', 2), ('E', 3)],

    'B': [('A', 2), ('C', 1), ('G', 9)],

    'C': [('B', 1)],

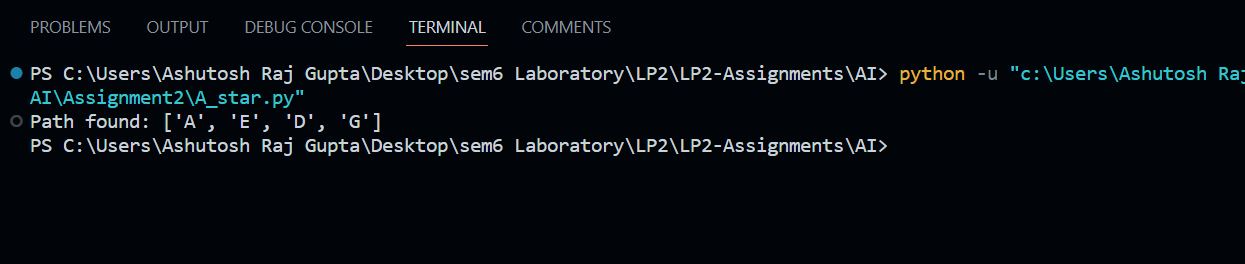
    'D': [('E', 6), ('G', 1)],

    'E': [('A', 3), ('D', 6)],

    'G': [('B', 9), ('D', 1)]

}

aStarAlgo('A', 'G')

****