Q1.**WRITE AN PYTHONPROGRAM FOR 8 PUZZLE PROBLEM**

AIM = an python program for 8 puzzle problem

ALGORITHM =

Step 1. State Representation: Represent the puzzle as a 3x3 grid, where each cell contains a number (1-8) or is empty (0).

step 2. node Representation: Each node in the search tree represents a state of the puzzle. It contains the current state, the previous state (parent), the move that led to this state, and the cost (usually the sum of the path cost and a heuristic estimate).

Step 3. Heuristic Function: Choose a heuristic function that estimates the cost from the current state to the goal state. A common heuristic for the 8-puzzle is the Manhattan distance, which is the sum of the horizontal and vertical distances of each tile to its correct position.

Step 4. Priority Queue: Use a priority queue (e.g., a min-heap) to store nodes during the search. Nodes are dequeued based on their total cost (path cost + heuristic cost).

PROGRAM =

import copy

from heapq import heappush, heappop

n = 3

rows = [ 1, 0, -1, 0 ]

cols = [ 0, -1, 0, 1 ]

class priorityQueue:

def \_init\_(self):

self.heap = []

def push(self, key):

heappush(self.heap, key)

def pop(self):

return heappop(self.heap)

def empty(self):

if not self.heap:

return True

else:

return False

class nodes:

def \_init\_(self, parent, mats, empty\_tile\_posi,

costs, levels):

self.parent = parent

self.mats = mats

self.empty\_tile\_posi = empty\_tile\_posi

self.costs = costs

self.levels = levels

def \_lt\_(self, nxt):

return self.costs < nxt.costs

def calculateCosts(mats, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mats[i][j]) and

(mats[i][j] != final[i][j])):

count += 1

return count

def newNodes(mats, empty\_tile\_posi, new\_empty\_tile\_posi,

levels, parent, final) -> nodes:

new\_mats = copy.deepcopy(mats)

x1 = empty\_tile\_posi[0]

y1 = empty\_tile\_posi[1]

x2 = new\_empty\_tile\_posi[0]

y2 = new\_empty\_tile\_posi[1]

new\_mats[x1][y1], new\_mats[x2][y2] = new\_mats[x2][y2], new\_mats[x1][y1]

costs = calculateCosts(new\_mats, final)

new\_nodes = nodes(parent, new\_mats, new\_empty\_tile\_posi,

costs, levels)

return new\_nodes

def printMatsrix(mats):

for i in range(n):

for j in range(n):

print("%d " % (mats[i][j]), end = " ")

print()

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatsrix(root.mats)

print()

def solve(initial, empty\_tile\_posi, final):

pq = priorityQueue()

costs = calculateCosts(initial, final)

root = nodes(None, initial,

empty\_tile\_posi, costs, 0)

pq.push(root)

while not pq.empty():

minimum = pq.pop()

if minimum.costs == 0:

printPath(minimum)

return

for i in range(n):

new\_tile\_posi = [

minimum.empty\_tile\_posi[0] + rows[i],

minimum.empty\_tile\_posi[1] + cols[i], ]

if isSafe(new\_tile\_posi[0], new\_tile\_posi[1]):

child = newNodes(minimum.mats,

minimum.empty\_tile\_posi,

new\_tile\_posi,

minimum.levels + 1,

minimum, final,)

pq.push(child)

initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

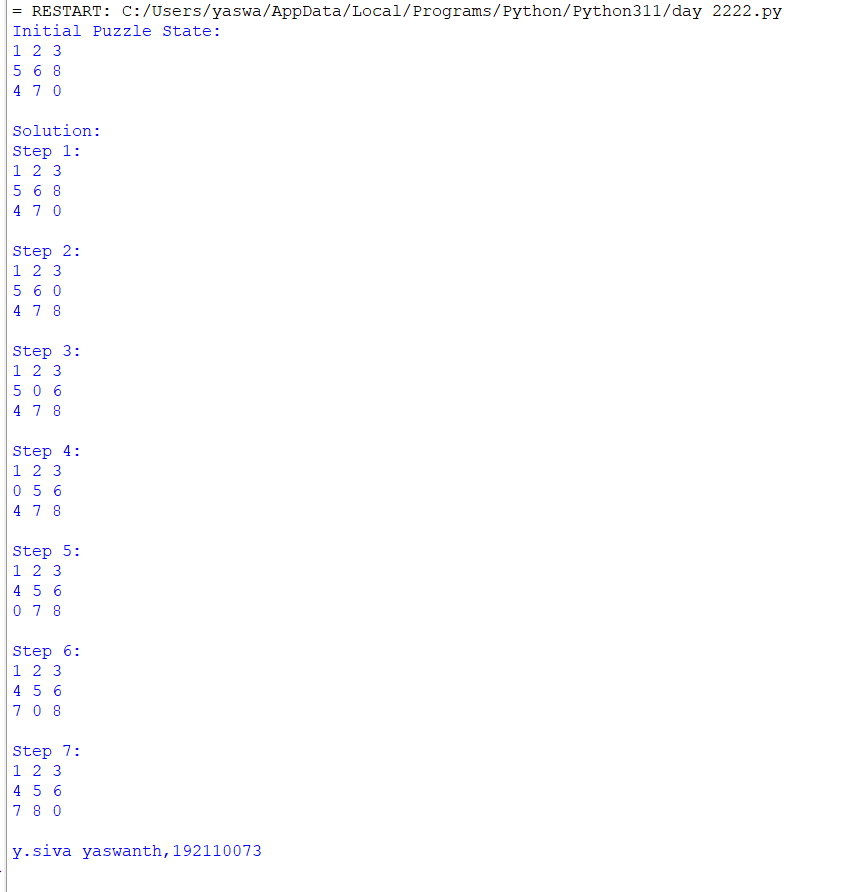
final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

empty\_tile\_posi = [ 1, 2 ]

solve(initial, empty\_tile\_posi, final)



RESULT= an python program for 8 puzzle problem run successfully

Q2. WRITE a python program for 8 queens problem

AIM= a python program for 8 queens problem

ALGORITHM =

STEP-1: Get the input for the user for n no.of queens to find .

STEP-2: Define a function attack(). Which is used to check that whether is there any

Queen placed on horizontal or in straight line.

STEP-3: Use another function n\_queens(). To place the queens in the row by row after

Placing one-another.

STEP-4: Use for-loop to iterate the loop and check the queen's position.

STEP-5: Print the loop.

PROGRAM =

print ("Enter the number of queens")

N = int(input())

# here we create a chessboard

# NxN matrix with all elements set to 0

board = [[0]\*N for \_ in range(N)]

def attack(i, j):

#checking vertically and horizontally

for k in range(0,N):

if board[i][k]==1 or board[k][j]==1:

return True

#checking diagonally

for k in range(0,N):

for l in range(0,N):

if (k+l==i+j) or (k-l==i-j):

if board[k][l]==1:

return True

return False

def N\_queens(n):

if n==0:

return True

for i in range(0,N):

for j in range(0,N):

if (not(attack(i,j))) and (board[i][j]!=1):

board[i][j] = 1

if N\_queens(n-1)==True:

return True

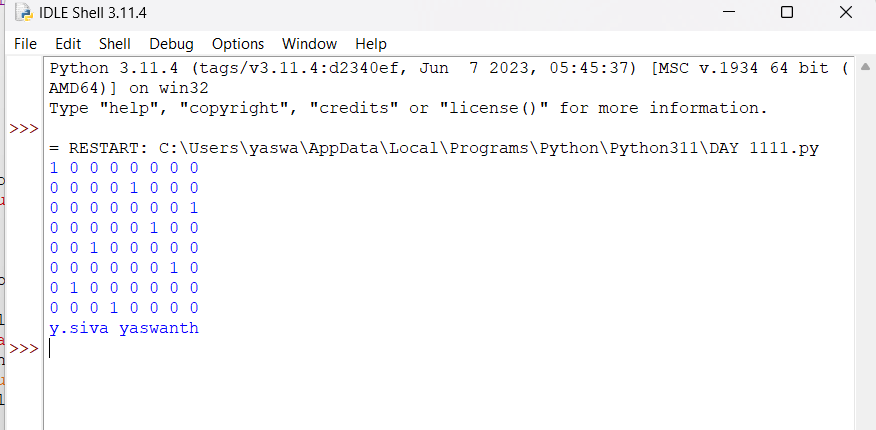
board[i][j] = 0

return False

N\_queens(N)

for i in board:

print (i)



RESULT=a python program for 8 queens problem ran successfully

3.Python program for vacuum cleaner problem

AIM = python program for vacuum cleaner problem

ALGORITHM

STEPS:-

1. Initialize vacuum cleaner's position at starting point

2. Initialize a list to keep track of cleaned locations

3. while there are uncleaned locations in the room:

4. if current location is dirty:

5. Clean the current location

6. Add current location to the list of cleaned location

7. if there are uncleaned locations adjacent to the current location:

8. Move to the closest uncleaned location

PROGRAM =

class VacuumCleaner:

def \_\_init\_\_(self):

self.position = 0 # Position of the vacuum cleaner (0: Left, 1: Right)

self.environment = [0, 0] # Environment with dirty cells (0: Clean, 1: Dirty)

def sense(self):

return self.environment[self.position]

def move(self):

if self.position == 0:

self.position = 1

else:

self.position = 0

def clean(self):

self.environment[self.position] = 0

def run(self, steps):

for \_ in range(steps):

current\_state = self.sense()

if current\_state == 1: # Dirty cell

self.clean()

print(f"Cleaned cell {self.position}")

else:

print(f"Cell {self.position} is already clean.")

self.move()

if \_\_name\_\_ == "\_\_main\_\_":

vacuum\_cleaner = VacuumCleaner()

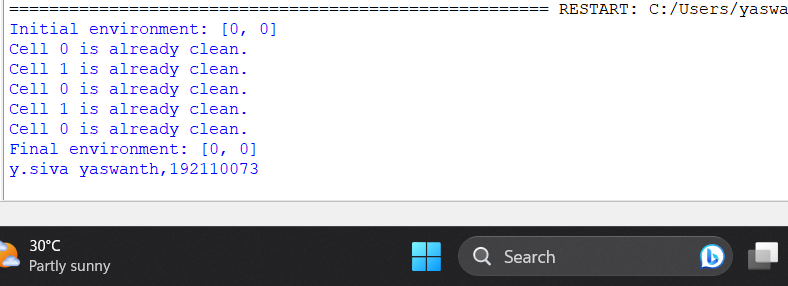
steps = 5

print("Initial environment:", vacuum\_cleaner.environment)

vacuum\_cleaner.run(steps)

print("Final environment:", vacuum\_cleaner.environment)

OUTPUT =



RESULT=python program for vacuum cleaner problem ran successfully.

4.PYTHON PROGRAM FOR SOLVING TOWERS OF HANOI PROBLEMS

AIM= python program for solving towers of hanoi problems

ALGORITHM =

STEPS:-

1.To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1

2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

1. First, we move the smaller (top) disk to aux peg.
2. Then, we move the larger (bottom) disk to destination peg.
3. And finally, we move the smaller disk from aux to destination peg.

PROGRAM =

def hanoi(n, source, auxiliary, destination):

if n == 1:

print(f"Move disk 1 from {source} to {destination}")

return

hanoi(n-1, source, destination, auxiliary)

print(f"Move disk {n} from {source} to {destination}")

hanoi(n-1, auxiliary, source, destination)

if \_\_name\_\_ == "\_\_main\_\_":

num\_disks = 3

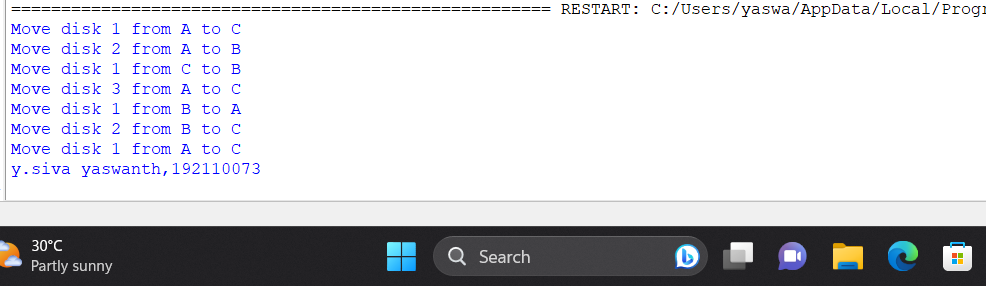
source\_peg = "A"

auxiliary\_peg = "B"

destination\_peg = "C"

hanoi(num\_disks, source\_peg, auxiliary\_peg, destination\_peg)

OUTPUT =



RESULT=python program for solving towers of hanoi problems ran successfully.

5.PYTHON PROGRAM FOR BFS AND DFS

AIM=python program for bfs and dfs

ALGORITHM =

BFS:

STEP-1:-Initialize a queue and add the initial state to it.

STEP-2:-While the queue is not empty:

STEP-3:- Dequeue a state from the front of the queue.

STEP-4:- Check if the state is the goal state. If yes, the solution is found.

Generate all possible successor states from the current state.

STEP-5:- Enqueue the successor states at the back of the queue.

If the queue becomes empty without finding the goal state, the search has failed

DFS:

STEP-1:-Initialize a stack and add the initial state to it.While the stack is not empty:

Pop a state from the top of the stack.

STEP-2:- Check if the state is the goal state. If yes, the solution is found.

STEP-3:- Generate all possible successor states from the current state.

STEP-4:- Push the successor states onto the stack.

If the stack becomes empty without finding the goal state, the search has failed

PROGRAM =

from collections import defaultdict

# Function to perform BFS

def bfs(graph, start):

visited = set()

queue = [start]

visited.add(start)

while queue:

node = queue.pop(0)

print(node, end=" ")

for neighbor in graph[node]:

if neighbor not in visited:

queue.append(neighbor)

visited.add(neighbor)

# Function to perform DFS

def dfs(graph, start):

visited = set()

stack = [start]

while stack:

node = stack.pop()

if node not in visited:

print(node, end=" ")

visited.add(node)

stack.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

if \_\_name\_\_ == "\_\_main\_\_":

# Example graph represented as an adjacency list

graph = defaultdict(list)

graph[0] = [1, 2]

graph[1] = [2]

graph[2] = [0, 3]

graph[3] = [3]

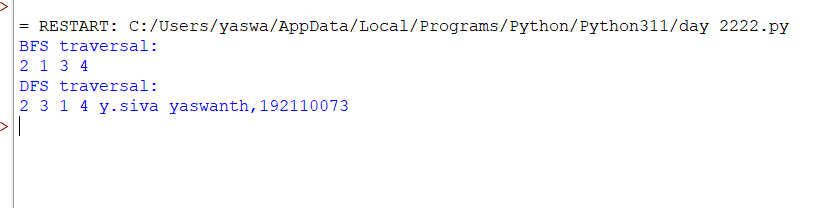
print("BFS traversal:")

bfs(graph, 2)

print("\nDFS traversal:")

dfs(graph, 2)

OUTPUT =



RESULT=python program for bfs and dfs ran successfully.

**Q6. Write the python program for Map Coloring**

**AIM:** Python program for Map coloring

ALGORITHM:

STEP-1:-Create a recursive function that takes the graph, current index, number of vertices, and output color array.

STEP-2:-If the current index is equal to the number of vertices. Print the color configuration in the output array.

STEP-3:-Assign a color to a vertex (1 to m).

STEP-4:-For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with the next index and number of vertices

PROGRAM=

def is\_safe(graph, node, color, coloring):

for neighbor in graph[node]:

if neighbor in coloring and coloring[neighbor] == color:

return False

return True

def map\_coloring(graph, colors, coloring, nodes):

if not nodes:

return True

node = nodes[0]

for color in colors:

if is\_safe(graph, node, color, coloring):

coloring[node] = color

if map\_coloring(graph, colors, coloring, nodes[1:]):

return True

coloring.pop(node, None)

return False

def main():

# Define the graph as an adjacency list

graph = {

'A': ['B', 'C'],

'B': ['A', 'C', 'D'],

'C': ['A', 'B', 'D', 'E'],

'D': ['B', 'C', 'E', 'F'],

'E': ['C', 'D'],

'F': ['D']

}

# Define the available colors

colors = ['Red', 'Green', 'Blue', 'Yellow']

# Initialize an empty coloring dictionary

coloring = {}

# Start coloring the map for all nodes in the graph

nodes = list(graph.keys())

if map\_coloring(graph, colors, coloring, nodes):

print("Map coloring solution:")

for node, color in coloring.items():

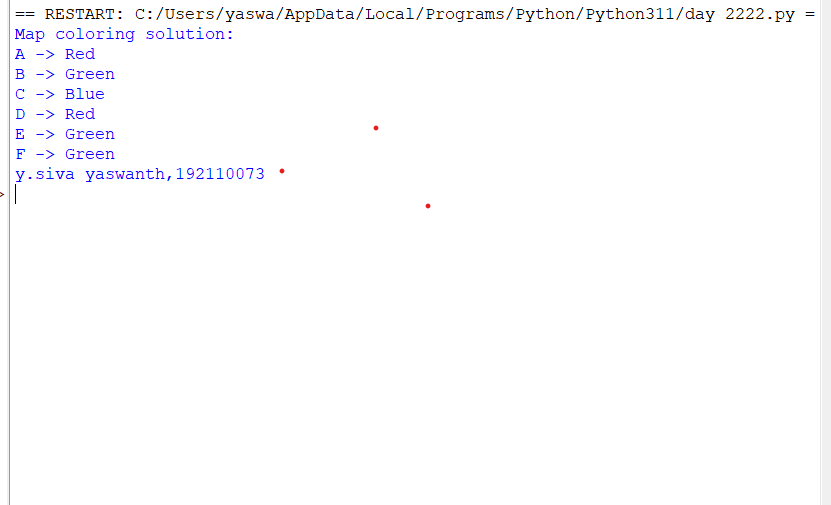
print(f"{node} -> {color}")

else:

print("No valid coloring found for the map.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

****

**RESULT=**Python program for Map coloring ran successfully.

**Q7.1.** **Prolog program for forward chaining and backward chaining**

**ALGORITHM=**

**PROGRAM=**

**OUTPUT=**

Q8.**Write the python program to implement A\* algorithm**

AIM=the python programming of A\* algorithm

ALGORITHM=

Steps:-

1. Firstly, Place the starting node into OPEN and find its f (n) value.
2. Then remove the node from OPEN, having the smallest f (n) value. If it is a goal node, then stop and return to success.
3. Else remove the node from OPEN, and find all its successors.
4. Find the f (n) value of all the successors, place them into OPEN, and place the removed node into close.

PROGRAM=

import heapq

def heuristic(node, goal):

# Replace this with an appropriate heuristic function

return 0

def astar(graph, start, goal):

open\_list = [(0, start)]

came\_from = {}

g\_score = {node: float('inf') for node in graph}

g\_score[start] = 0

while open\_list:

current\_cost, current\_node = heapq.heappop(open\_list)

if current\_node == goal:

path = []

while current\_node in came\_from:

path.insert(0, current\_node)

current\_node = came\_from[current\_node]

path.insert(0, start)

return path

for neighbor, cost in graph[current\_node].items():

tentative\_g\_score = g\_score[current\_node] + cost

if tentative\_g\_score < g\_score[neighbor]:

came\_from[neighbor] = current\_node

g\_score[neighbor] = tentative\_g\_score

f\_score = tentative\_g\_score + heuristic(neighbor, goal)

heapq.heappush(open\_list, (f\_score, neighbor))

return None

# Example usage:

graph = {

'A': {'B': 1, 'C': 3},

'B': {'A': 1, 'C': 1, 'D': 4},

'C': {'A': 3, 'B': 1, 'D': 1},

'D': {'B': 4, 'C': 1},

}

start\_node = 'A'

goal\_node = 'D'

path = astar(graph, start\_node, goal\_node)

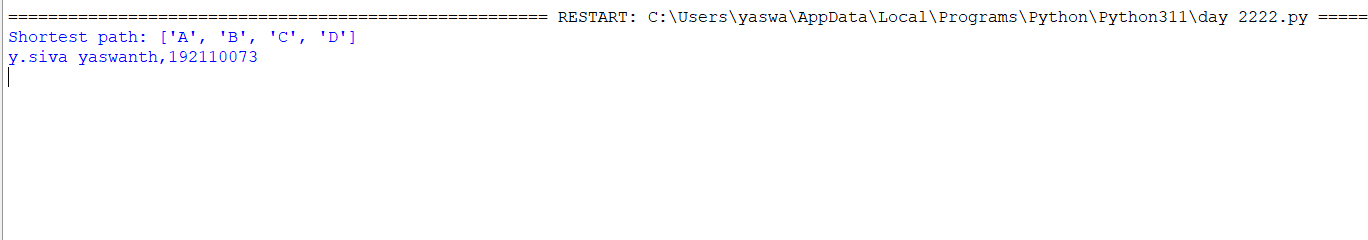
if path:

print("Shortest path:", path)

else:

print("No path found.")

OUTPUT=



RESULT=the python programming of A\* algorithm ran successfully.

Q9.WRITE THE PYTHON PROGRAM FOR TRAVELLING SALESMEN PROBLEM

AIM=Python program for travelling salesmen problem

ALGORITHM=

Steps:-

1. Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G’) which will record the path the salesman is going to take from one node to another.
2. The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
3. The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
4. Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
5. Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.

PROGRAM=

import itertools

def calculate\_distance(points, order):

total\_distance = 0

num\_points = len(order)

for i in range(num\_points):

point1 = points[order[i]]

point2 = points[order[(i + 1) % num\_points]]

total\_distance += calculate\_distance\_between\_points(point1, point2)

return total\_distance

def calculate\_distance\_between\_points(point1, point2):

return ((point1[0] - point2[0])\*\*2 + (point1[1] - point2[1])\*\*2) \*\* 0.5

def tsp\_bruteforce(points):

min\_distance = float('inf')

optimal\_order = None

num\_points = len(points)

for order in itertools.permutations(range(num\_points)):

distance = calculate\_distance(points, order)

if distance < min\_distance:

min\_distance = distance

optimal\_order = order

return min\_distance, optimal\_order

# Example usage:

points = {

'A': (0, 0),

'B': (1, 2),

'C': (4, 1),

'D': (2, 4)

}

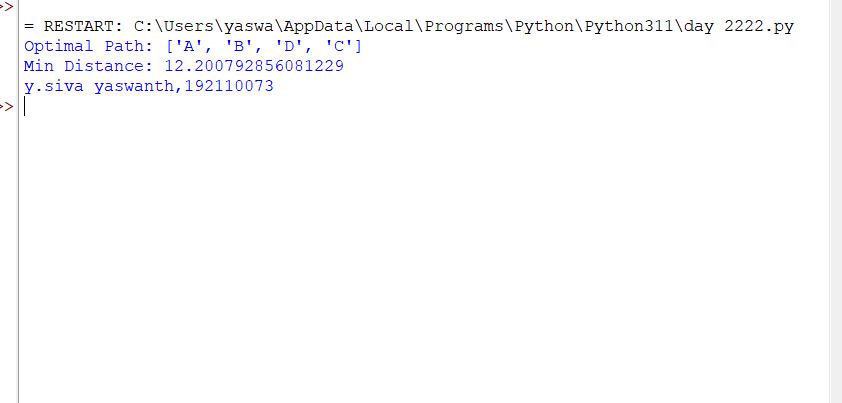
min\_distance, optimal\_order = tsp\_bruteforce(list(points.values()))

optimal\_path = [list(points.keys())[i] for i in optimal\_order]

print("Optimal Path:", optimal\_path)

print("Min Distance:", min\_distance)

output=



RESULT=Python program for travelling salesmen problem ran successfully.

Q10.WRITE A PYTHON PROGRAM FOR GREEDY BEST FIRST SEARCH

AIM=a python program for greedy best first search.

ALGORITHM=

Steps:-

1. Initialize a tree with the root node being the start node in the open list.
2. If the open list is empty, return a failure, otherwise, add the current node to the closed list.
3. Remove the node with the lowest h(x) value from the open list for exploration.
4. If a child node is the target, return a success. Otherwise, if the node has not been in either the open or closed list, add it to the open list for exploration.

PROGRAM=

import heapq

def heuristic(node, goal):

# Replace this with an appropriate heuristic function

return 0

def greedy\_best\_first\_search(graph, start, goal):

open\_list = [(heuristic(start, goal), start)]

closed\_set = set()

while open\_list:

\_, current\_node = heapq.heappop(open\_list)

if current\_node == goal:

return True # Goal reached

closed\_set.add(current\_node)

for neighbor, \_ in graph[current\_node].items():

if neighbor not in closed\_set:

heapq.heappush(open\_list, (heuristic(neighbor, goal), neighbor))

return False # Goal not reachable

# Example usage:

graph = {

'A': {'B': 5, 'C': 2},

'B': {'A': 5, 'D': 4},

'C': {'A': 2, 'D': 7},

'D': {'B': 4, 'C': 7, 'E': 3},

'E': {'D': 3, 'F': 1},

'F': {'E': 1}

}

start\_node = 'A'

goal\_node = 'F'

result = greedy\_best\_first\_search(graph, start\_node, goal\_node)

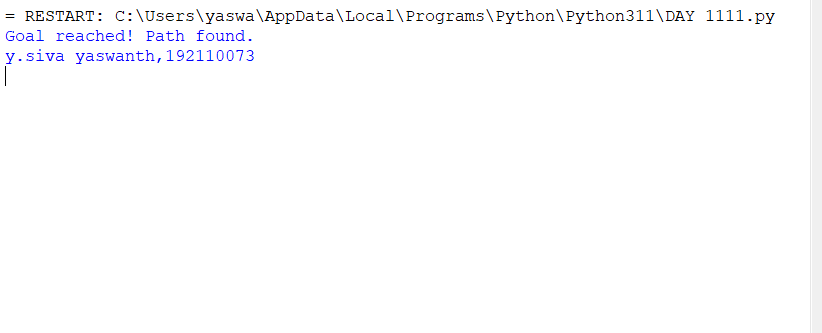
if result:

print("Goal reached! Path found.")

else:

print("Goal not reachable.")

OUTPUT=



RESULT=a python program for greedy best first search ran successfully.