**Name: Sahil Babar**

**Roll No: 75**

**Class: TE CO**

**Task 1 : Assignment (DFS & BFS)**

**Code:**

**1)** **DFS**

**import java.io.\*;**

**import java.util.\*;**

**class Graph {**

**private int V;**

**private LinkedList<Integer> adj[];**

**@SuppressWarnings("unchecked") Graph(int v)**

**{**

**V = v;**

**adj = new LinkedList[v];**

**for (int i = 0; i < v; ++i)**

**adj[i] = new LinkedList();**

**}**

**void addEdge(int v, int w)**

**{**

**adj[v].add(w);**

**}**

**void DFSUtil(int v, boolean visited[])**

**{**

**visited[v] = true;**

**System.out.print(v + " ");**

**Iterator<Integer> i = adj[v].listIterator();**

**while (i.hasNext()) {**

**int n = i.next();**

**if (!visited[n])**

**DFSUtil(n, visited);**

**}**

**}**

**void DFS(int v)**

**{**

**boolean visited[] = new boolean[V];**

**DFSUtil(v, visited);**

**}**

**public static void main(String args[])**

**{**

**Graph g = new Graph(4);**

**g.addEdge(0, 1);**

**g.addEdge(0, 2);**

**g.addEdge(1, 2);**

**g.addEdge(2, 0);**

**g.addEdge(2, 3);**

**g.addEdge(3, 3);**

**System.out.println(**

**"Following is Depth First Traversal "**

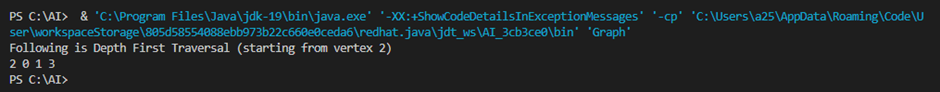
**+ "(starting from vertex 2)");**

**g.DFS(2);**

**}**

**}**

**Output:**

****

**2)** **BFS Code:**

**import java.io.\*;**

**import java.util.\*;**

**class Graph1 {**

**private int V;**

**private LinkedList<Integer> adj[];**

**Graph1(int v)**

**{**

**V = v;**

**adj = new LinkedList[v];**

**for (int i = 0; i < v; ++i)**

**adj[i] = new LinkedList();**

**}**

**void addEdge(int v, int w) { adj[v].add(w); }**

**void BFS(int s)**

**{**

**boolean visited[] = new boolean[V];**

**LinkedList<Integer> queue**

**= new LinkedList<Integer>();**

**visited[s] = true;**

**queue.add(s);**

**while (queue.size() != 0) {**

**s = queue.poll();**

**System.out.print(s + " ");**

**Iterator<Integer> i = adj[s].listIterator();**

**while (i.hasNext()) {**

**int n = i.next();**

**if (!visited[n]) {**

**visited[n] = true;**

**queue.add(n);**

**}**

**}**

**}**

**}**

**public static void main(String args[])**

**{**

**Graph1 g = new Graph1(4);**

**g.addEdge(0, 1);**

**g.addEdge(0, 2);**

**g.addEdge(1, 2);**

**g.addEdge(2, 0);**

**g.addEdge(2, 3);**

**g.addEdge(3, 3);**

**System.out.println(**

**"Following is Breadth First Traversal "**

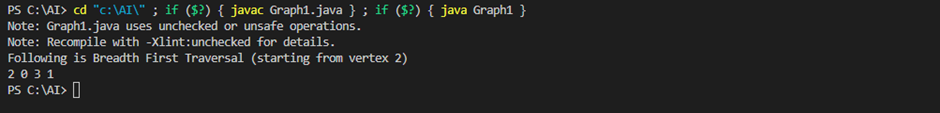
**+ "(starting from vertex 2)");**

**g.BFS(2);**

**}**

**}**

**TASK 1 Output:**



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

**Code:**

**class Graph:**

**def \_\_init\_\_(self, adjacency\_list):**

**self.adjacency\_list = adjacency\_list**

**def get\_neighbors(self, v):**

**return self.adjacency\_list[v]**

**def h(self, n):**

**H = {**

**'A': 11,**

**'B': 6,**

**'C': 99,**

**'D': 1,**

**'E': 7,**

**'G': 0**

**}**

**return H[n]**

**def a\_star\_algorithm(self, start\_node, stop\_node):**

**open\_list = set([start\_node])**

**closed\_list = set([])**

**g = {}**

**g[start\_node] = 0**

**parents = {}**

**parents[start\_node] = start\_node**

**while len(open\_list) > 0:**

**n = None**

**for v in open\_list:**

**if n == None or g[v] + self.h(v) < g[n] + self.h(n):**

**n = v;**

**if n == None:**

**print('Path does not exist!')**

**return None**

**if n == stop\_node:**

**reconst\_path = []**

**while parents[n] != n:**

**reconst\_path.append(n)**

**n = parents[n]**

**reconst\_path.append(start\_node)**

**reconst\_path.reverse()**

**print('Path found: {}'.format(reconst\_path))**

**return reconst\_path**

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

**if m not in open\_list and m not in closed\_list:**

**open\_list.add(m)**

**parents[m] = n**

**g[m] = g[n] + weight**

**else:**

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

**g[m] = g[n] + weight**

**parents[m] = n**

**if m in closed\_list:**

**closed\_list.remove(m)**

**open\_list.add(m)**

**open\_list.remove(n)**

**closed\_list.add(n)**

**print('Path does not exist!')**

**return None**

**adjac\_lis = {**

**'A': [('B', 2), ('E', 3)],**

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

**'C': None,**

**'D': [('G', 1)],**

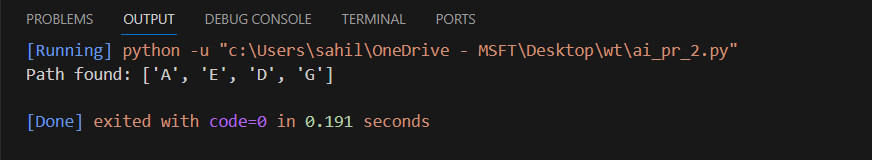
**'E': [('D', 6)]**

**}**

**graph = Graph(adjac\_lis)**

**graph.a\_star\_algorithm('A', 'G')**

**OutPut**

****

**TASK 3 Implement greedy search for following application**

**Code**

**Selection sort**

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min\_index = i

for j in range(i + 1, n):

if arr[j] < arr[min\_index]:

min\_index = j

arr[i], arr[min\_index] = arr[min\_index], arr[i]

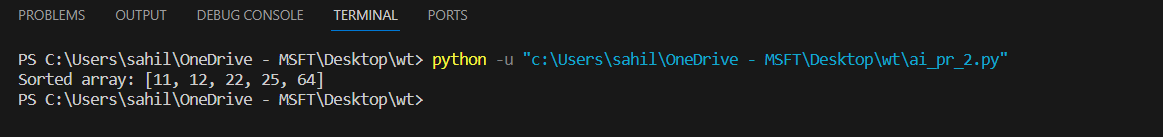
return arr

arr = [64, 25, 12, 22, 11]

sorted\_arr = selection\_sort(arr)

print("Sorted array:", sorted\_arr)

**Output**

****

II. Minimum Spanning Tree

import heapq

def prim(graph):

# Initialize an empty list to store the minimum spanning tree

mst = []

# Choose any arbitrary starting vertex

start\_vertex = list(graph.keys())[0]

# Use a priority queue to keep track of the edges with the minimum weight

priority\_queue = [(0, start\_vertex, None)]

# Set to keep track of visited vertices

visited = set()

while priority\_queue:

# Get the edge with the minimum weight

weight, current\_vertex, previous\_vertex = heapq.heappop(priority\_queue)

# Check if the current vertex has been visited

if current\_vertex not in visited:

# Mark the current vertex as visited

visited.add(current\_vertex)

# Add the edge to the minimum spanning tree

if previous\_vertex is not None:

mst.append((previous\_vertex, current\_vertex, weight))

# Explore the neighbors of the current vertex

for neighbor, neighbor\_weight in graph[current\_vertex]:

heapq.heappush(priority\_queue, (neighbor\_weight, neighbor, current\_vertex))

return mst

# Example usage:

# Define a graph as an adjacency list

example\_graph = {

'A': [('B', 2), ('D', 5)],

'B': [('A', 2), ('C', 3), ('D', 1)],

'C': [('B', 3), ('D', 4)],

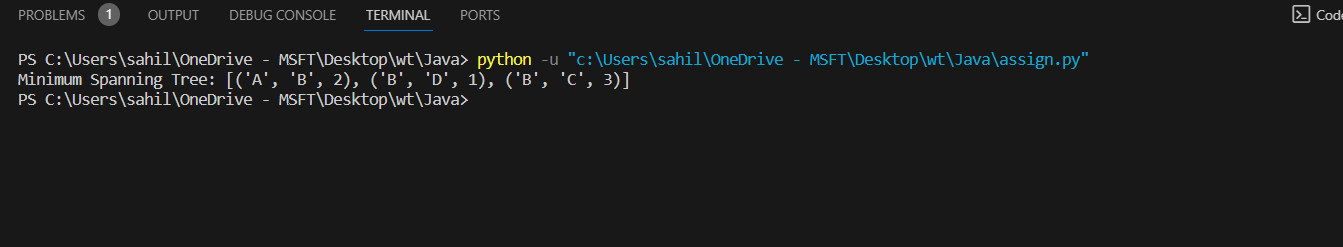
'D': [('A', 5), ('B', 1), ('C', 4)]

}

result = prim(example\_graph)

print("Minimum Spanning Tree:", result)

Output



III. Single-Source Shortest Path Problem

Code

import heapq

def dijkstra(graph, start):

# Initialize a dictionary to store the distances from the start vertex

distances = {vertex: float('infinity') for vertex in graph}

# Set the distance from the start vertex to itself to 0

distances[start] = 0

# Use a priority queue to keep track of vertices with the minimum distance

priority\_queue = [(0, start)]

while priority\_queue:

current\_distance, current\_vertex = heapq.heappop(priority\_queue)

# Check if the current distance is smaller than the stored distance

if current\_distance > distances[current\_vertex]:

continue

# Explore neighbors of the current vertex

for neighbor, weight in graph[current\_vertex]:

distance = current\_distance + weight

# If the new distance is smaller, update the distance and push to the queue

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

return distances

# Example usage:

# Define a graph as an adjacency list

example\_graph = {

'A': [('B', 2), ('D', 5)],

'B': [('A', 2), ('C', 3), ('D', 1)],

'C': [('B', 3), ('D', 4)],

'D': [('A', 5), ('B', 1), ('C', 4)]

}

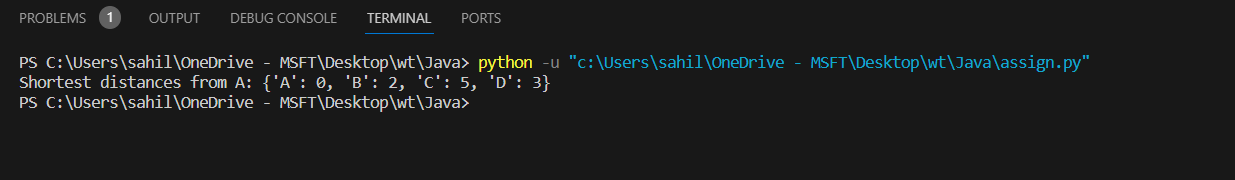
# Specify the start vertex

start\_vertex = 'A'

result = dijkstra(example\_graph, start\_vertex)

print(f"Shortest distances from {start\_vertex}: {result}")

Output



IV. Job Scheduling Problem

def job\_scheduling(jobs):

# Sort jobs by their finish times

jobs.sort(key=lambda x: x[1])

# Initialize an array to store the maximum weight at each index

max\_weight = [0] \* len(jobs)

for i in range(len(jobs)):

# Find the latest non-overlapping job

latest\_non\_overlapping = binary\_search(jobs, i)

# Calculate the maximum weight at the current index

incl\_weight = jobs[i][2]

if latest\_non\_overlapping != -1:

incl\_weight += max\_weight[latest\_non\_overlapping]

# Store the maximum weight at the current index

max\_weight[i] = max(incl\_weight, max\_weight[i - 1] if i - 1 >= 0 else 0)

return max\_weight[-1]

def binary\_search(jobs, index):

# Find the latest non-overlapping job using binary search

low, high = 0, index - 1

while low <= high:

mid = (low + high) // 2

if jobs[mid][1] <= jobs[index][0]:

if jobs[mid + 1][1] <= jobs[index][0]:

low = mid + 1

else:

return mid

else:

high = mid - 1

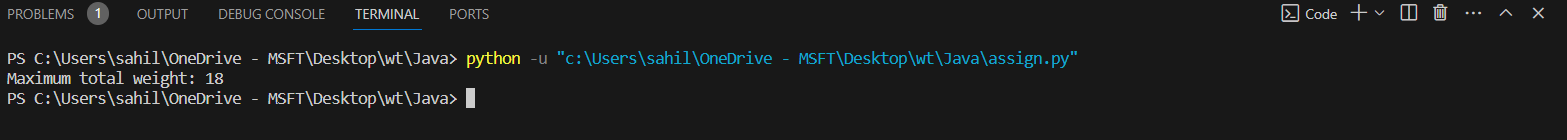
return -1

jobs = [(1, 3, 5), (2, 5, 6), (6, 8, 8), (4, 6, 5)]

result = job\_scheduling(jobs)

print("Maximum total weight:", result)

Output



V. Prim's Minimal Spanning Tree Algorithm

Code

import heapq

def prim(graph):

# Initialize an empty set to store visited vertices

visited = set()

# Choose any arbitrary starting vertex

start\_vertex = list(graph.keys())[0]

# Use a priority queue to keep track of the edges with the minimum weight

priority\_queue = [(0, start\_vertex, None)]

# Initialize an empty list to store the edges of the minimal spanning tree

mst\_edges = []

while priority\_queue:

# Get the edge with the minimum weight

weight, current\_vertex, previous\_vertex = heapq.heappop(priority\_queue)

# Check if the current vertex has been visited

if current\_vertex not in visited:

# Mark the current vertex as visited

visited.add(current\_vertex)

# Add the edge to the minimal spanning tree

if previous\_vertex is not None:

mst\_edges.append((previous\_vertex, current\_vertex, weight))

# Explore the neighbors of the current vertex

for neighbor, neighbor\_weight in graph[current\_vertex]:

heapq.heappush(priority\_queue, (neighbor\_weight, neighbor, current\_vertex))

return mst\_edges

# Example usage:

# Define a graph as an adjacency list

example\_graph = {

'A': [('B', 2), ('D', 5)],

'B': [('A', 2), ('C', 3), ('D', 1)],

'C': [('B', 3), ('D', 4)],

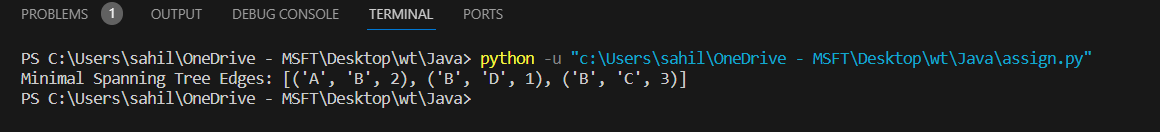
'D': [('A', 5), ('B', 1), ('C', 4)]

}

result = prim(example\_graph)

print("Minimal Spanning Tree Edges:", result)

Output



VI. Kruskal's Minimal Spanning Tree Algorithm

def kruskal\_mst(graph):

graph.sort(key=lambda edge: edge[2])

parent = {i: i for i in range(len(graph))}

result = []

def find(u):

if parent[u] != u:

parent[u] = find(parent[u])

return parent[u]

def union(u, v):

root\_u, root\_v = find(u), find(v)

parent[root\_u] = root\_v

for u, v, w in graph:

if find(u) != find(v):

result.append((u, v, w))

union(u, v)

return result

# Example usage:

graph = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

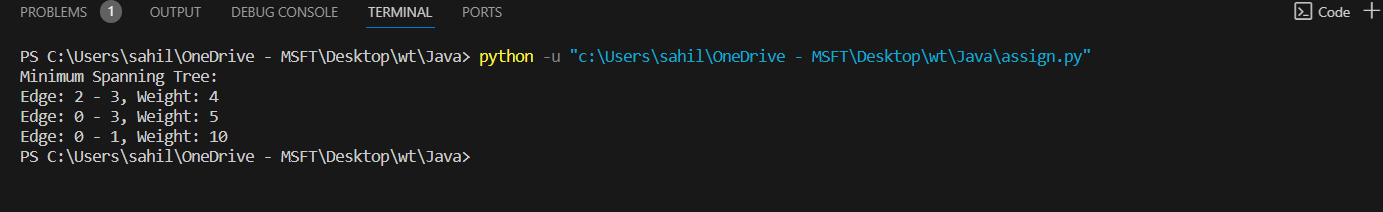
mst = kruskal\_mst(graph)

print("Minimum Spanning Tree:")

for u, v, w in mst:

print(f"Edge: {u} - {v}, Weight: {w}")

Output



VII. Dijkstra's Minimal Spanning Tree Algorithm

Code

import heapq

def dijkstra(graph, start):

distances = {vertex: float('infinity') for vertex in graph}

distances[start] = 0

priority\_queue = [(0, start)]

while priority\_queue:

current\_distance, current\_vertex = heapq.heappop(priority\_queue)

if current\_distance > distances[current\_vertex]:

continue

for neighbor, weight in graph[current\_vertex]:

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

return distances

# Example usage:

# Represent the graph as an adjacency list with weights

graph = {

'A': [('B', 1), ('C', 4)],

'B': [('A', 1), ('C', 2), ('D', 5)],

'C': [('A', 4), ('B', 2), ('D', 1)],

'D': [('B', 5), ('C', 1)]

}

start\_vertex = 'A'

shortest\_paths = dijkstra(graph, start\_vertex)

print(f"Shortest paths from vertex {start\_vertex}:")

for vertex, distance in shortest\_paths.items():

print(f"To {vertex}: {distance}")

Output

