- Problem 3:

```
def Dijkstra_algorithm(self, start, end):
    # Initialize the distances to all vertices as infinity
    distances = {vertex: float('inf') for vertex in self.graph}
    # The distance to the start vertex is 0
    distances[start] = 0
    # Initialize an empty dictionary to store the previous vertex on the
shortest path
    previous = {}
    # Create a priority queue to store vertices that need to be visited
    queue = [(0, start)]
    # While the queue is not empty
    while queue:
        # Get the vertex with the smallest distance from the start vertex
        current_distance, current_vertex = heapq.heappop(queue)
        # If we've already visited the vertex, skip it
        if current_distance > distances[current_vertex]:
            continue
            # For each neighbor of the current vertex
        for edge in self.graph[current_vertex]:
            neighbor = edge[0]
            weight = edge[1]
            # Calculate the distance to the neighbor
            distance = current_distance + weight
            # If the distance is less than the current distance to the
neighbor
            if distance < distances[neighbor]:
                # Update the distance to the neighbor
                distances[neighbor] = distance
                # Set the previous vertex of the neighbor to the current
vertex
                previous[neighbor] = current_vertex
                # Add the neighbor to the priority queue
                heapq.heappush(queue, (distance, neighbor))
        # If there is no path from the start vertex to the end vertex,
return None
    if end not in previous:
        return None
        # Initialize an empty list to store the path from the start vertex
to the end vertex
    path = []
    # Start at the end vertex and work backwards to the start vertex
    current_vertex = end
    while current_vertex != start:
        path.append(current_vertex)
        current_vertex = previous[current_vertex]
    path.append(start)

    # Reverse the path to get it in the correct order
    path.reverse()
    # Return the lowest cost walk and its cost
    return path, distances[end]
```

➔ Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph. For its implementation, we initialise a dictionary for distances, one that

holds the previous nodes of the current ones, and a prority queue. While the queue isn't empty, we pop its contents and calculate the distance. As we progress, if we find smaller distances than the ones we previously did, they get updated, along with previous. After all, we initialize an empty list to store the path from the start vertex to the end vertex (we have it in previous). We append evertyhing, reverse it, and return it, along with the end's distance(the final distance of everything).