Title ***(English)***	OSPF(bare bone) Phase 1 : Dijkstra algorithm
Team	Marina Reda Abdullah Mekhael 221101235 Omar adly mahmoud nasr 221101398 Hamza Nashaat Abdelbaki 221100328 Mohamed Ahmed Fathi 221101699 Aly Maher Abdelfattah 221101789 Abdelrahman Mohamed Mahmoud 221101107 Sohila ahmed zakria 221101149
Field	Computer science and Engineering
Program	Artificial intelligence science
Instructor	**Dr. Mohamed Amir**
This part for the instructor: Notice	**Degree**
Date of Submission	

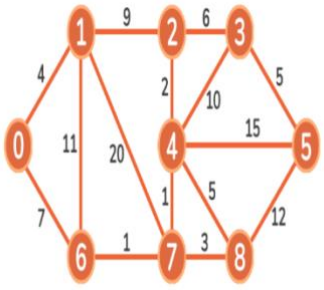
Section One: Introduction to Dijkstra's Algorithm

Dijkstra's algorithm is a search strategy devised for computing the shortest path between a selected node (often termed as the 'source') and all other nodes in a graph characterized by non-negative edge weights. Its significance is underlined by its extensive utility in numerous practical applications, such as routing algorithms for telecommunications, traffic navigation systems, and in artificial intelligence for pathfinding in games. The computational complexity of Dijkstra's algorithm is significantly influenced by the data structures used for the priority queue. Implementations with a simple array have a time complexity of O(n^2), while those with a binary heap (or Fibonacci heap) reduce this to O((V+E)logV), where V is the number of vertices and E is the number of edges. It is often preferred for its efficiency over the Bellman-Ford algorithm, which, despite being applicable to graphs with negative weights, operates at a slower O(VE) complexity. Unlike the A* algorithm, which uses heuristics to optimize the search towards a single target node and can often find the shortest path faster on graphs with a vast number of nodes, Dijkstra's algorithm is comprehensive, calculating the shortest path to all nodes without the use of heuristics.

Section Two: Explanation of Dijkstra's Algorithm Mechanism

1. **Initialization**: The algorithm begins by setting the distance to the source node as zero and all other nodes as infinity, indicating that they are initially unreachable from the source. A priority queue is then created and initialized with the source node.
2. **Priority Queue Operations**: The priority queue, which orders nodes by their current shortest distance, is a pivotal structure in the algorithm. It ensures that the node with the least tentative distance is always processed next.
3. **Relaxation**: As the algorithm proceeds, it relaxes the edges by examining all the unvisited neighbors of the current node. For each neighbor, the algorithm checks if the sum of the current distance and the edge weight is less than the known distance to the neighbor. If it is, the algorithm updates the neighbor's distance to this smaller value and notes the current node as a predecessor for this path.
4. **Extraction and Visitation**: The node with the smallest distance is extracted from the priority queue, and its distance is finalized as the shortest distance from the source. This node is then marked as visited, and it will not be processed again.
5. **Pathfinding**: Along with updating distances, the algorithm keeps a record of the path taken to reach each node by tracking the predecessor of each node. This allows for the reconstruction of the shortest path by backtracking from the destination node to the source node.
6. **Termination**: The algorithm repeats the extraction, relaxation, and visitation steps until the priority queue is empty, indicating that the shortest path to every reachable node has been determined.

Section Three: Result Section with Test Case



-The user is prompted to enter the number of nodes and edges in the graph. In this case, the graph has 9 nodes and 15 edges.
- Each edge of the graph is entered by specifying the start node, end node, and the weight of the edge.
- The user is prompted to enter the start node for the shortest path calculations, which is node 0 in this instance.

Destination	Cost	Next Hop
0	0	-
1	4	0
2	11	4
3	17	2
4	9	7
5	22	3
6	7	0
7	8	6
8	11	7

ex : node 2 has next hop 4
node 4 has next hop 7
node 7 has next hop 6
node 6 has next hop 0
this is path to node 2 [0,6,7,4,2]

```
Enter the number of nodes: 9
Enter the number of edges: 15
Enter each edge in the format: start_node end_node weight
Enter edge (e.g., 0 1 4): 0 1 4
Enter edge (e.g., 0 1 4): 0 6 7
Enter edge (e.g., 0 1 4): 1 6 11
Enter edge (e.g., 0 1 4): 1 7 20
Enter edge (e.g., 0 1 4): 1 2 9
Enter edge (e.g., 0 1 4): 6 7 1
Enter edge (e.g., 0 1 4): 4 7 1
Enter edge (e.g., 0 1 4): 2 4 2
Enter edge (e.g., 0 1 4): 2 3 6
Enter edge (e.g., 0 1 4): 7 8 3
Enter edge (e.g., 0 1 4): 4 3 10
Enter edge (e.g., 0 1 4): 4 8 5
Enter edge (e.g., 0 1 4): 4 5 15
Enter edge (e.g., 0 1 4): 3 5 5
Enter edge (e.g., 0 1 4): 5 8 12
Enter the start node: 0
Shortest distances from node 0
Distance to node 0: 0, Path: [0]
Distance to node 1: 4, Path: [0, 1]
Distance to node 2: 11, Path: [0, 6, 7, 4, 2]
Distance to node 3: 17, Path: [0, 6, 7, 4, 2, 3]
Distance to node 4: 9, Path: [0, 6, 7, 4]
Distance to node 5: 22, Path: [0, 6, 7, 4, 2, 3, 5]
Distance to node 6: 7, Path: [0, 6]
Distance to node 7: 8, Path: [0, 6, 7]
Distance to node 8: 11, Path: [0, 6, 7, 8]
```

It outlines the shortest path from a starting node (node 0) to all other nodes, detailing both the minimum distance and the exact path to each node. For instance, the shortest path to node 2 is of length 11, through the route [0, 6, 7, 4, 2]. This execution trace is a classic illustration of the algorithm's application in finding the most efficient routes in a weighted graph.

Vertex	Distance	Path	Link Identifier
0	0	[0]	_
1	4	[0, 1]	1
2	11	[0, 6, 7, 4, 2]	2
3	17	[0, 6, 7, 4, 2, 3]	2
4	9	[0, 6, 7, 4]	2
5	22	[0, 6, 7, 4, 2, 3, 5]	2
6	7	[0, 6]	2
7	8	[0, 6, 7]	2
8	11	[0, 6, 7, 8]	2

Appendix section:

```
] import heapq  # Importing the heapq module for priority queue operations

  # Define the main function for Dijkstra's algorithm
  def dijkstra_with_paths(num_nodes, edges, start_vertex):
      distances = {v: float('infinity') for v in range(num_nodes)}
      distances[start_vertex] = 0
      previous_nodes = {v: None for v in range(num_nodes)}
      pq = []
      heapq.heappush(pq, (0, start_vertex))
      visited = set()

      while pq:
          dist, current_vertex = heapq.heappop(pq)
          if current_vertex in visited:
              continue
          visited.add(current_vertex)

          for neighbor, weight in edges[current_vertex]:
              if neighbor not in visited:
                  old_cost = distances[neighbor]
                  new_cost = dist + weight
                  if new_cost < old_cost:
                      distances[neighbor] = new_cost
                      previous_nodes[neighbor] = current_vertex
                      heapq.heappush(pq, (new_cost, neighbor))

      return distances, previous_nodes

  # Function to reconstruct the shortest path from start_vertex to end_vertex
  # and generate a link identifier for the path
  def reconstruct_path(previous_nodes, end_vertex):
      path = []
      current_vertex = end_vertex
      while current_vertex is not None:
          path.append(current_vertex)
          current_vertex = previous_nodes[current_vertex]
      path.reverse()
      # Check if the path has at least two nodes to get the second node in the path
      link_identifier = path[1] if len(path) > 1 else "Direct"
      return path, link_identifier

  def print_results_in_table_format(distances, previous_nodes, num_nodes):
      print(f"{'Vertex':<10} {'Distance':<15} {'Path':<30} {'Link Identifier'}")
      second_node_to_identifier = {}  # Maps the second node in the path to an identifier
      identifier_counter = 1  # Start numbering identifiers from 1

      for end_vertex in range(num_nodes):
          path, _ = reconstruct_path(previous_nodes, end_vertex)

          # Check if the path has at least two nodes to get the second node in the path
          if len(path) > 1:
              second_node = path[1]
              if second_node not in second_node_to_identifier:
                  second_node_to_identifier[second_node] = identifier_counter
                  identifier_counter += 1
              link_identifier = second_node_to_identifier[second_node]
          else:
              link_identifier = '_'

          cost = distances[end_vertex] if distances[end_vertex] != float('infinity') else '∞'
          print(f"{end_vertex:<10} {cost:<15} {str(path):<30} {link_identifier}")
```

```
# Main function to take user inputs and execute the algorithm
def dijkstra_input():
    num_nodes = int(input("Enter the number of nodes: "))
    num_edges = int(input("Enter the number of edges: "))

    edges = {i: [] for i in range(num_nodes)}
    print("Enter each edge in the format: start_node end_node weight")
    for _ in range(num_edges):
        start, end, weight = map(int, input("Enter edge (e.g., 0 1 4): ").split())
        edges[start].append((end, weight))
        edges[end].append((start, weight))  # If undirected graph

    start_vertex = int(input("Enter the start node: "))

    distances, previous_nodes = dijkstra_with_paths(num_nodes, edges, start_vertex)
    print_results_in_table_format(distances, previous_nodes, num_nodes)

# Check if the script is run as the main program
if __name__ == "__main__":
    dijkstra_input()
```