23CCE201-Data Structures **CB.EN.U4CCE23005**

Shortest Path Algorithms for Weighted Directed Graphs

Aim:

To understand and implement various shortest path algorithms for a weighted directed graph.

Logic:

The weighted directed graphs have vertices and edges joining vertices with cost on them. The goal is to find the shortest path, that is the path between vertices where the sum of cost of edges is minimum.

There are three different algorithms that can help us achieve the solution for this problem:

- Dijkstra's algorithm:

  It helps us find the shortest path from a single source vertex to all other vertices in a graph.

- Dynamic Programming:

  It also helps us find the shortest path from a single source vertex to all vertices in a graph, but the Dijkstra's algorithm is based on greedy approach whereas dynamic program achieves the same result with iterative approach till the shortest path is found

- Floyd's algorithm:

  It helps in finding the shortest paths between all pairs of vertices.

Algorithm:

Dijkstra's algorithm: (graph, start)

```
1) Start
2) Initialize a dictionary distance to store the distances from source vertex “start” to other vertices. Initialize all the distances to be infinite and the distance from source vertex to source vertex to be zero.
3) Initialize a min heap with a tuple in it for source vertex. Tuple format (distance, vertex)
4)Pop from the min heap and store the data as current_distance and current_vertex
5) If the current_distance is greater than distance of  current_vertex then moves to step 4.
6) In a loop of all vertices in the graph:
    6.1) Initialize a variable distance to store the alternate distance where distance will be sum of distance of current_vertex and cost of edge in iterator.
    6.2)If distance < distance of vertex in iterator then move to step 6.2.1.
        6.2.1) Initialize distance of vertex in iterator to distance
        6.2.2) Push the tuple (distance ,vertex in iterator) to the min heap.
7) If min heap is not empty then move to step 4.
8) End.
```

Dynamic Programming:

```
1) Start
2) Initialize Distances from Source:
    2.1) Create a dictionary distances to store the minimum distance from the source to
    each vertex.
    2.2) Set all distances of vertices to infinity .
    2.3) Set the distance of the source vertex to 0.

3) Relax Edges |V|−1 Times:

    3.1) For each vertex in the graph, repeat the following /V/–1|V| - 1|V|−1 times:
        3.1.1) For each vertex u in the graph:
            3.1.1.1) For each neighbor v of u with an edge cost weight:
                    3.1.1.1.1) If the distance to u is not infinity and
                    distances[u] + weight < distances[v], update
                    distances[v] to distances[u] + weight.

4) Check for Negative-Weight Cycles:

    4.1) For each vertex u in the graph:
        4.1.1) For each neighbor v with edge weight weight:
            4.1.1.1) If distances[u] + weight < distances[v],return None else return
            Distances.

5)End
```

Floyd's algorithm: (graph)

```
  1. Start
  2. Initialize Distance Matrix:
        2.1) Create a 2D list dist to store the shortest distances between all pairs of
        vertices.
        2.2) Set all values in dist to infinity (float('inf')) initially.
  3. Set Initial Distances Based on Graph:
        3.1) For each vertex u:
            3.1.1) For each vertex v:
                3.1.1.1) Set dist[u][v] to the value of graph[u][v] if there's an
                    edge; otherwise, set it to infinity.
                3.1.1.2) Set the distance from each vertex to itself (dist[u][u])
                    to 0.
  4. Iterate Over Intermediate Vertices:
        4.1) For each vertex k (acting as an intermediate vertex):
            4.1.1) For each vertex i:
                4.1.1.1) For each vertex j:
                    4.1.1.1.1) Update dist[i][j] to the minimum of its current
                    value and the path passing through k: dist[i][j] =
                    min(dist[i][j], dist[i][k] + dist[k][j]).
5. End
```

<u>Code:</u>

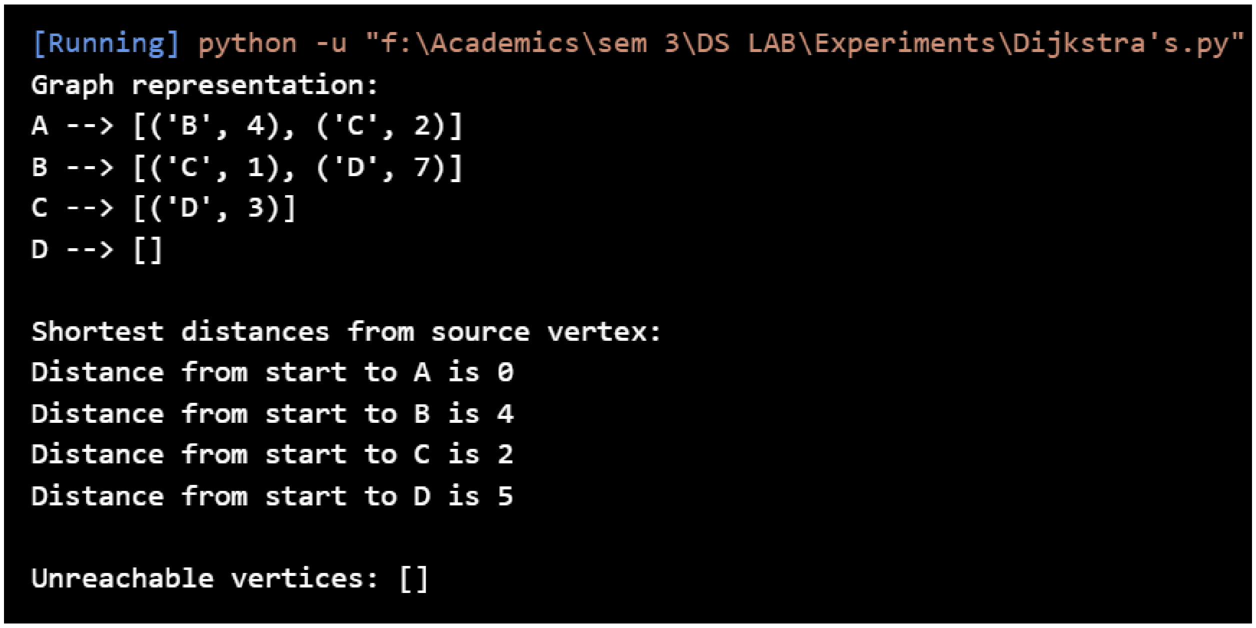
Dijkstra:

```
class Graph:

  def __init__(self, adjacency_matrix):

    self.graph = {}

    self.initialize_graph(adjacency_matrix)


  # Initialize the graph with an adjacency matrix

  def initialize_graph(self, adjacency_matrix):

    for vertex_1 in adjacency_matrix:

      self.graph[vertex_1] = []

      for vertex_2, cost in adjacency_matrix[vertex_1].items():

        if cost != float('inf'):  # Skip if there's no direct edge

          self.graph[vertex_1].append((vertex_2, cost))


  # Print the graph

  def print_graph(self):
    for vertex, edges in self.graph.items():
        print(f"{vertex} --> {edges}")


# Print distances
def print_distances(self, distances):
    for vertex, distance in distances.items():
        print(f"Distance from start to {vertex} is {distance}")
class MinHeap:
    def __init__(self):
        self.heap = []


    def push(self, element):
        self.heap.append(element)
        self._heapify_up(len(self.heap) - 1)


    def pop(self):
        if len(self.heap) == 0:
            return None
        self._swap(0, len(self.heap) - 1)
        min_element = self.heap.pop()
        self._heapify_down(0)
        return min_element
```

```
def _heapify_up(self, index):

    parent_index = (index - 1) // 2

    if index > 0 and self.heap[index][0] < self.heap[parent_index][0]:

        self._swap(index, parent_index)

        self._heapify_up(parent_index)


def _heapify_down(self, index):

    smallest = index

    left_child = 2 * index + 1

    right_child = 2 * index + 2


    if left_child < len(self.heap) and self.heap[left_child][0] < self.heap[smallest][0]:

        smallest = left_child

    if right_child < len(self.heap) and self.heap[right_child][0] < self.heap[smallest][0]:

        smallest = right_child


    if smallest != index:

        self._swap(index, smallest)

        self._heapify_down(smallest)


def _swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]


    def is_empty(self):
        return len(self.heap) == 0


# Dijkstra's algorithm
def dijkstra(self, start):
    min_heap = self.MinHeap()
    min_heap.push((0, start))  # (distance, vertex)
    distances = {vertex: float("infinity") for vertex in self.graph}
    distances[start] = 0
    visited = set()  # To track visited vertices


    while not min_heap.is_empty():
        current_distance, current_vertex = min_heap.pop()


        if current_vertex in visited:
            continue
        visited.add(current_vertex)


        # Explore all neighbors of the current vertex
        for neighbor, weight in self.graph[current_vertex]:
}

            distance = current_distance + weight

            # Only consider this new path if it's better
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                min_heap.push((distance, neighbor))

        # Find unvisited vertices (unreachable vertices)
        unreachable = [vertex for vertex in self.graph if vertex not in visited]

        return distances, unreachable


# Example usage with adjacency matrix
if __name__ == "__main__":
    # Adjacency matrix represented as a dictionary of dictionaries with alphabetic legends
   adjacency_matrix = {
    'A': {'B': 4, 'C': 2, 'D': float('inf')},
    'B': {'C': 1, 'D': 7},
    'C': {'D': 3},
    'D': {}
}
```

```
new_graph = Graph(adjacency_matrix)

print("Graph representation:")

new_graph.print_graph()

 start_vertex = 'A'

distances, unreachable = new_graph.dijkstra(start_vertex)

print("\nShortest distances from source vertex:")

new_graph.print_distances(distances)

print("\nUnreachable vertices:", unreachable)
```

Result:



```
[Running] python -u "f:\Academics\sem 3\DS LAB\Experiments\Dijkstra's.py"
Graph representation:
A --> [('B', 4), ('C', 2)]
B --> [('C', 1), ('D', 7)]
C --> [('D', 3)]
D --> []

Shortest distances from source vertex:
Distance from start to A is 0
Distance from start to B is 4
Distance from start to C is 2
Distance from start to D is 5

Unreachable vertices: []
```

Inference:

This graph implementation uses an adjacency matrix to define and visualize vertex connections and edge weights. Dijkstra's algorithm, supported by a min-heap, efficiently calculates the shortest paths and identifies unreachable vertices from a starting point.

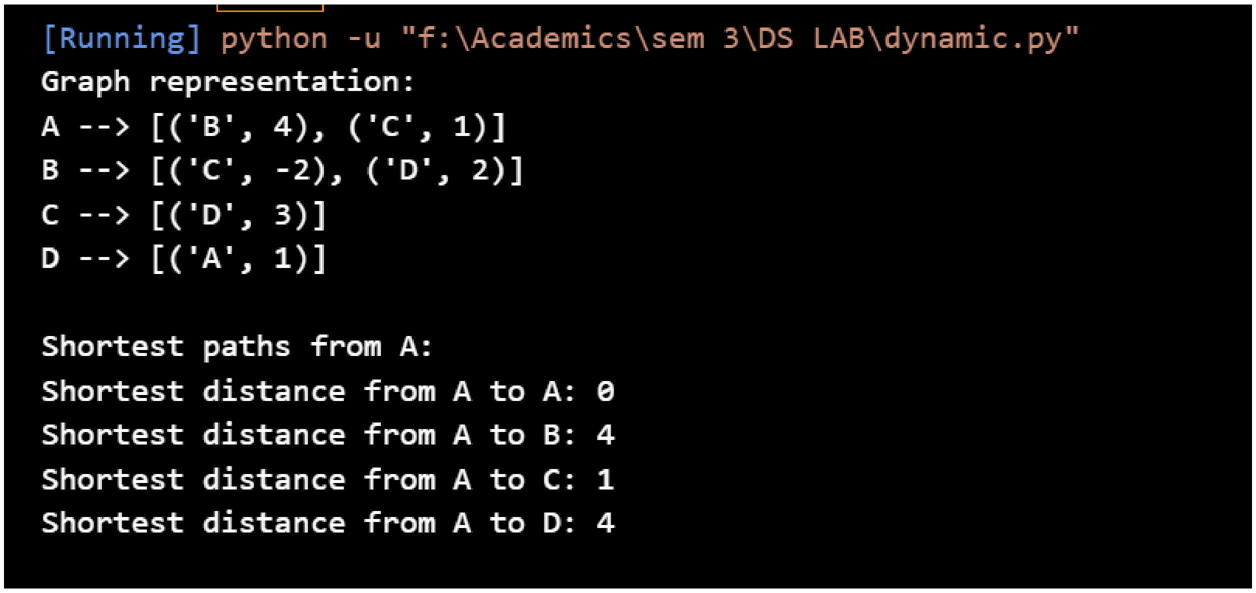
Code:

Dynamic programming:

```
class Graph:

  def __init__(self, adjacency_matrix):

    self.graph = {}

    self.initialize_graph(adjacency_matrix)


  # Initialize the graph with an adjacency matrix

  def initialize_graph(self, adjacency_matrix):

    for vertex_1 in adjacency_matrix:

      self.graph[vertex_1] = []

      for vertex_2, cost in adjacency_matrix[vertex_1].items():

        if cost != float('inf'):  # Skip if there's no direct edge

          self.graph[vertex_1].append((vertex_2, cost))


  # Print the graph

  def print_graph(self):

    for vertex, edges in self.graph.items():

      print(f"{vertex} --> {edges}")


  # Print shortest paths from the source to each vertex

  def print_shortest_paths(self, source, distances):
        for vertex, dist in distances.items():
            print(f"Shortest distance from {source} to {vertex}: {dist}")

    # Bellman-Ford algorithm for shortest path from a single source
    def bellman_ford(self, source):
        # Initialize distances from the source
        distances = {vertex: float('inf') for vertex in self.graph}
        distances[source] = 0

        # Relax edges |V| - 1 times
        for _ in range(len(self.graph) - 1):
            for vertex in self.graph:
                for neighbor, weight in self.graph[vertex]:
                    if distances[vertex] != float('inf') and distances[vertex] + weight <
distances[neighbor]:
                        distances[neighbor] = distances[vertex] + weight

        # Check for negative-weight cycles
        for vertex in self.graph:
            for neighbor, weight in self.graph[vertex]:
                if distances[vertex] != float('inf') and distances[vertex] + weight <
distances[neighbor]:
                    print("Graph contains a negative-weight cycle")
                    return None

        return distances
```

```
# Example usage with adjacency matrix
if __name__ == "__main__":
    # Adjacency matrix represented as a dictionary of dictionaries
    adjacency_matrix = {
        'A': {'B': 4, 'C': 2, 'D': float('inf')},
        'B': {'C': 1, 'D': 7},
        'C': {'D': 3},
        'D': {}
    }

    new_graph = Graph(adjacency_matrix)

    print("Graph representation:")
    new_graph.print_graph()

    source = 'A'
    distances = new_graph.bellman_ford(source)

    if distances:
        print(f"\nShortest paths from {source}:")
        new_graph.print_shortest_paths(source, distances)
```

Result:



```
[Running] python -u "f:\Academics\sem 3\DS LAB\dynamic.py"
Graph representation:
A --> [('B', 4), ('C', 1)]
B --> [('C', -2), ('D', 2)]
C --> [('D', 3)]
D --> [('A', 1)]

Shortest paths from A:
Shortest distance from A to A: 0
Shortest distance from A to B: 4
Shortest distance from A to C: 1
Shortest distance from A to D: 4
```

<u>Inference:</u>

Unlike Dijkstra’s algorithm Dynamic programming helps us find shortest path of directed graphs with negative costs.

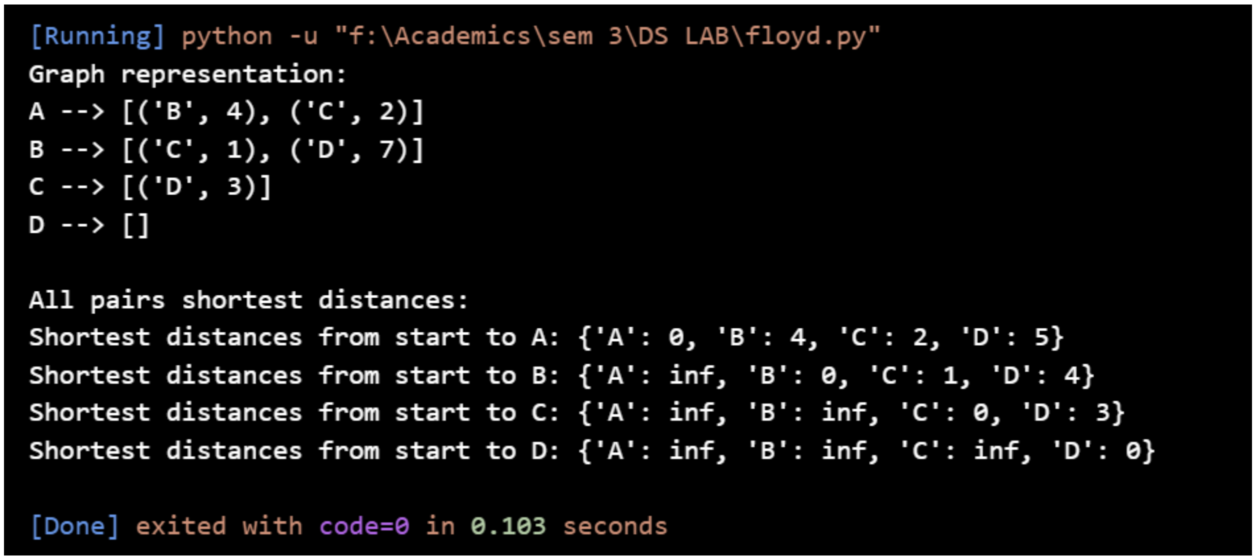
<u>Code:</u>

Floyd’s Algorithm:

```
class Graph:
  def __init__(self, adjacency_matrix):
    self.graph = {}
    self.initialize_graph(adjacency_matrix)


  # Initialize the graph with an adjacency matrix
  def initialize_graph(self, adjacency_matrix):
    for vertex_1 in adjacency_matrix:
      self.graph[vertex_1] = []
      for vertex_2, cost in adjacency_matrix[vertex_1].items():
        if cost != float('inf'):  # Skip if there's no direct edge
            self.graph[vertex_1].append((vertex_2, cost))

# Print the graph
def print_graph(self):
    for vertex, edges in self.graph.items():
        print(f"{vertex} --> {edges}")

# Print all pairs shortest paths
def print_all_pairs_shortest_paths(self, distances):
    for vertex, dist in distances.items():
        print(f"Shortest distances from start to {vertex}: {dist}")

# Floyd-Warshall's algorithm
def floyd_warshall(self):
    vertices = list(self.graph.keys())
    num_vertices = len(vertices)

    # Create a distance matrix initialized to the adjacency matrix
    distances = {vertex: {v: float('inf') for v in vertices} for vertex in vertices}
    for vertex in vertices:
        distances[vertex][vertex] = 0  # Distance to self is zero
        for neighbor, weight in self.graph[vertex]:
            distances[vertex][neighbor] = weight

        # Update the distances based on the Floyd-Warshall algorithm
        for k in vertices:
            for i in vertices:
                for j in vertices:
                    if distances[i][j] > distances[i][k] + distances[k][j]:
                        distances[i][j] = distances[i][k] + distances[k][j]

        return distances


# Example usage with adjacency matrix
if __name__ == "__main__":
    # Adjacency matrix represented as a dictionary of dictionaries
    adjacency_matrix = {
    'A': {'B': 4, 'C': 2, 'D': float('inf')},
    'B': {'C': 1, 'D': 7},
    'C': {'D': 3},
    'D': {}
}

    new_graph = Graph(adjacency_matrix)
```

```
print("Graph representation:")

new_graph.print_graph()


distances = new_graph.floyd_warshall()


print("\nAll pairs shortest distances:")

new_graph.print_all_pairs_shortest_paths(distances)
```

Result:



```
[Running] python -u "f:\Academics\sem 3\DS LAB\floyd.py"
Graph representation:
A --> [('B', 4), ('C', 2)]
B --> [('C', 1), ('D', 7)]
C --> [('D', 3)]
D --> []

All pairs shortest distances:
Shortest distances from start to A: {'A': 0, 'B': 4, 'C': 2, 'D': 5}
Shortest distances from start to B: {'A': inf, 'B': 0, 'C': 1, 'D': 4}
Shortest distances from start to C: {'A': inf, 'B': inf, 'C': 0, 'D': 3}
Shortest distances from start to D: {'A': inf, 'B': inf, 'C': inf, 'D': 0}

[Done] exited with code=0 in 0.103 seconds
```

Inference:

Floyd's algorithm finds the shortest paths between all pairs of vertices, even with negative edge weights, by progressively considering each vertex as an intermediate step in paths.