May 1, 2024

Department of Computer Science and Engineering
Islamic University of Technology (IUT)
A subsidiary organ of OIC

Algorithm Engineering Lab 03
Azmayen Fayek Sabil, 190042122

Contents

1 Task 1

1.1 Python Code for Finding Minimum Weight Cycle in Undirected Graph

To find the minimum weighted cycle in an undirected graph, we iterate through each edge, temporarily remove it, calculate the shortest path between its vertices using Dijkstra's algorithm, add the weight of the edge to this path to form a cycle, and update the minimum weighted cycle encountered so far. This process is repeated for each edge, and the smallest weighted cycle found is returned as the result.

1.1.1 Edge Class

This class represents an edge in the graph. It stores the vertices and weight of the edge.

```python
class Edge:

    def __init__(self, u: int, v: int, weight: int) -> None:
        self.u = u
        self.v = v
        self.weight = weight
```

Listing 1: Edge class

1.1.2 Graph Class

This class represents the graph. It has methods to add and remove edges, find the shortest path using Dijkstra's algorithm, and find the minimum weighted cycle.

```python
class Graph:

    def __init__(self, V: int) -> None:
        self.V = V
        self.adj = [[] for _ in range(V)]
        self.edge = []

    def addEdge(self, u: int, v: int, w: int) -> None:
        self.adj[u].append((v, w))
        self.adj[v].append((u, w))
        e = Edge(u, v, w)
        self.edge.append(e)

    # Other methods...
```

Listing 2: Graph class

1.1.3 ShortestPath Method

This method calculates the shortest path between two vertices `u` and `v` in the graph using Dijkstra's shortest path algorithm.

```python
 def ShortestPath(self, u: int, v: int) -> int:
        setds = set()
        dist = [INF] * self.V
        setds.add((0, u))
        dist[u] = 0
        while (setds):
            tmp = setds.pop()
            uu = tmp[1]
            for i in self.adj[uu]:
                vv = i[0]
                weight = i[1]
                if (dist[vv] > dist[uu] + weight):
                    if (dist[vv] != INF):
                        if ((dist[vv], vv) in setds):
                            setds.remove((dist[vv], vv))
                    dist[vv] = dist[uu] + weight
                    setds.add((dist[vv], vv))
        return dist[v]
```

Listing 3: Explanation of `ShortestPath` method

1.1.4 FindMinimumCycle Method

This method finds the minimum weighted cycle in the graph.

```python
def FindMinimumCycle(self) -> int:
        min_cycle = maxsize
        E = len(self.edge)
        for i in range(E):
            e = self.edge[i]
            self.removeEdge(e.u, e.v, e.weight)
            distance = self.ShortestPath(e.u, e.v)
            min_cycle = min(min_cycle, distance + e.weight)
            self.addEdge(e.u, e.v, e.weight)
        return min_cycle
```

Listing 4: Explanation of `FindMinimumCycle` method

1.1.5 Main Code

Creates a graph and adds edges to it. Then, it calls the `FindMinimumCycle` method to find the minimum weighted cycle in the graph.

```python
if __name__ == "__main__":
    V = 9
    g = Graph(V)
    g.addEdge(0, 1, 4)
    g.addEdge(0, 7, 8)
    g.addEdge(1, 2, 8)
    g.addEdge(1, 7, 11)
    g.addEdge(2, 3, 7)
    g.addEdge(2, 8, 2)
    g.addEdge(2, 5, 4)
    g.addEdge(3, 4, 9)
    g.addEdge(3, 5, 14)
    g.addEdge(4, 5, 10)
    g.addEdge(5, 6, 2)
    g.addEdge(6, 7, 1)
    g.addEdge(6, 8, 6)
    g.addEdge(7, 8, 7)
    print(g.FindMinimumCycle())
```

Listing 5: Main code

1.2 Complexity Analysis

1.2.1 Time Complexity

The time complexity of finding the minimum weighted cycle in the graph is $O(E^2 \log V)$. For each edge, we run Dijkstra's shortest path algorithm, which has a time complexity of $O(E \log V)$. Since there are E edges in the graph, the total time complexity is $O(E^2 \log V)$.

1.2.2 Space Complexity

The space complexity is $O(V^2)$ due to the adjacency list representation of the graph and other data structures used.

2 Task 2

2.1 Weird Country Roads

The Weird Country comprises n cities numbered from 0 to $n-1$, interconnected by r bidirectional roads. Each road has an associated cost c. The cost represents the expense (positive) or income (negative) incurred when traveling between adjacent cities. Mr. Exatiq resides in city 0 but has limited funds, with only m BDT (Bangladeshi Taka) remaining from his expenditures on RC cars. The task is to determine which cities he can afford to visit.

2.1.1 Input Format

The input consists of three integers: n (the number of cities), r (the number of roads), and m (Mr. Exatiq's budget). This is followed by r lines, each containing three integers u, v, and c, denoting a road between cities u and v with associated cost c.

2.1.2 Python Solution

```python
from collections import defaultdict

def can_visit_cities(n, r, m, roads):
    # We create an adjacency list to represent the graph
    graph = defaultdict(list)
    for u, v, c in roads:
        graph[u].append((v, c))
        graph[v].append((u, c))

    # We perform a DFS traversal to find reachable cities
    visited = set()
    stack = [0]  # We start from city 0
    while stack:
        current_city = stack.pop()
        visited.add(current_city)
        for neighbor, cost in graph[current_city]:
            if neighbor not in visited and cost <= m:
                stack.append(neighbor)

    return visited

# Example input
n, r, m = 5, 4, 6
roads = [(0, 1, 2), (0, 2, -4), (1, 3, 3), (3, 4, 2)]

# We find cities Mr. Exatiq can visit
reachable_cities = can_visit_cities(n, r, m, roads)
print(*reachable_cities)
```

Listing 6: Python solution to find reachable cities

2.1.3 Code Explanation

can_visit_cities function: This function takes four parameters: n (number of cities), r (number of roads), m (Mr. Exatiq's budget), and *roads* (a list of tuples representing roads between cities with associated costs). It creates an adjacency list representation of the graph using a defaultdict. Then, it performs a depth-first search (DFS) traversal starting from city 0 to find reachable cities within the budget.

2.1.4 Output Explanation

For the given example input, Mr. Exatiq can visit cities 0, 1, 2, 3, and 4. This is because he can afford to travel along roads with costs 2, -4, 3, and 2, respectively, while staying within his budget.

2.2 Complexity Analysis

2.2.1 Time Complexity

The time complexity of the solution is $O(r + n)$, where r is the number of roads and n is the number of cities. This complexity arises from building the adjacency list graph and performing the depth-first search (DFS) traversal.

2.2.2 Space Complexity

The space complexity of the solution is $O(r + n)$ due to the storage of the adjacency list graph and the visited set during DFS traversal.