CSCI 2300: Introduction to Algorithms

Homework 6

Lucien Brule
Prof. Bulent Yener
April 27, 2023

1 Problem 1

Problem: Given an undirected graph G, describe a linear time algorithm to find the number of distinct shortest paths between two given vertices u and v. Note that two shortest paths are distinct if they have at least one edge that is different.

Solution:

To find the number of distinct shortest paths between two vertices u and v, we can use a modified Breadth-First Search (BFS) algorithm. The algorithm is as follows:

1. Initialize an array ‘distance‘ of length equal to the number of vertices in the graph to store the shortest distance from u to each vertex. Set all elements to infinity except for ‘distance[u]‘, which is set to 0. 2. Initialize an array ‘count‘ of length equal to the number of vertices in the graph to store the number of distinct shortest paths from u to each vertex. Set all elements to 0 except for ‘count[u]‘, which is set to 1. 3. Create an empty queue an enqueue the starting vertex u. 4. While the queue is not empty, dequeue a vertex, and for each of its neighbors, check if the distance from u to the neighbor through the current vertex is less than the current recorded distance. If so, update the distance and set the count for the neighbor equal to the count of the current vertex. If the distance is equal to the current recorded distance, add the count of the current vertex to the count of the neighbor. 5. The number of distinct shortest paths between u and v is stored in ‘count[v]‘.

Here's the pseudo code for the algorithm:

```
function count_shortest_paths(adj_list, u, v):
    n = len(adj_list)
    distance = [float('inf')] * n
    count = [0] * n
    queue = []

    distance[u] = 0
    count[u] = 1
    queue.append(u)

    while queue:
        current_vertex = queue.pop(0)
        for neighbor in adj_list[current_vertex]:
            if distance[current_vertex] + 1 < distance[neighbor]:
```

```
                distance[neighbor] = distance[current_vertex] + 1
                count[neighbor] = count[current_vertex]
                queue.append(neighbor)
            elif distance[current_vertex] + 1 == distance[neighbor]:
                count[neighbor] += count[current_vertex]

    return count[v]
```

This algorithm has a time complexity of $O(|V| + |E|)$, as it performs a modified BFS traversal of the graph.

2 Problem 2

Problem: Given a weighted directed graph with positive weights, give an $O(|V|^3)$ algorithm to find the length of the shortest cycle or report that the graph is acyclic.

Solution:

To find the length of the shortest cycle in a weighted directed graph with positive weights, we can use the Floyd-Warshall algorithm with a slight modification. The algorithm is as follows:

1. Initialize a distance matrix 'dist' with dimensions $|V| \times |V|$ to store the shortest path distances between all pairs of vertices. Set the diagonal elements to 0, and the other elements to the corresponding edge weights or infinity if there is no edge between the vertices. 2. For each vertex k, update the distance matrix by checking if the distance from vertex i to vertex j through vertex k is shorter than the current recorded distance from vertex i to vertex j. If so, update the distance in the matrix. 3. Initialize a variable '$\min_{c}ycle_{l}ength$'$toinfinity.4.Iteratethroughthediagonalelementsoftheupdateddistancematrix, andu$

Here's the pseudo code for the algorithm:

```
function shortest_cycle_length(adj_matrix):
    n = len(adj_matrix)
    dist = adj_matrix.copy()

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

```
    min_cycle_length = float('inf')
    for i in range(n):
        min_cycle_length = min(min_cycle_length, dist[i][i])

    if min_cycle_length == float('inf'):
        return "The graph is acyclic."
    else:
        return min_cycle_length
```

This algorithm has a time complexity of $O(|V|^3)$, as it performs the Floyd-Warshall algorithm on the graph.

3 Problem 3

Problem: Given a directed weighted graph G, with positive weights on the edges, let us also add positive weights on the nodes. Let l(x,y) denote the weight of an edge (x,y), and let w(x) denote the weight of a vertex x. Define the cost of a path as the sum of the weights of all the edges and vertices on that path. Give an efficient algorithm to find all the smallest cost paths (as defined above) from a source vertex to all other vertices. Analyze and report the running time of your algorithm.

Solution:

We can modify Dijkstra's algorithm to solve this problem. We will update the relaxation step to consider the vertex weight in addition to the edge weight. Here's the modified algorithm:

```
1. Create a set of unvisited vertices.
2. Set the initial distance for the source vertex to its own weight,
    and set the distance for all other vertices to infinity.
3. While there are unvisited vertices:
    a. Select the vertex with the minimum distance value, mark it as visited,
        and call it the current vertex.
    b. For each neighbor of the current vertex:
        i. Calculate the cost of the path from the source vertex to the neighbor
            through the current vertex by summing the distance to the current
            vertex, the weight of the edge between the current vertex and
            the neighbor, and the weight of the neighbor.
        ii. If this cost is less than the current distance to the neighbor,
            update the distance for the neighbor.
```

```
4. Return the distances from the source vertex to all other vertices.
```

4 Problem 4

This is similar in implementation to the lab problem, reference the corresponding lab.