Lab Report: Johnson's Algorithm

Course: Algorithm Design and Analysis
Lab Title: Implementation of Johnson's Algorithm
Team Members:

- Imtiaz Ahmed (ID: 0432410005101138)
- Labib Ahmed (ID: 0432410005101133)
- Shykot Omi (ID: 0432410005101136)
- Fazle Rabbi (ID: 2125051075)

Objective

To understand and implement Johnson's Algorithm in C++ for finding shortest paths between all pairs of vertices in a sparse weighted directed graph. The algorithm is designed to handle graphs with negative weights.

Introduction

Johnson's Algorithm combines the Bellman-Ford and Dijkstra's algorithms to find shortest paths efficiently in sparse graphs. It reweights the edges using the Bellman-Ford algorithm to eliminate negative weights, and then uses Dijkstra's algorithm to compute the shortest paths for all pairs.

Algorithm Steps:

1. Add a new vertex, to the graph, connecting it to every other vertex with an edge of weight 0.
2. Use the Bellman-Ford algorithm starting from to compute the potential function h(v)h(v).
3. Reweight the edges of the graph using w'(u,v)=w(u,v)+h(u)−h(v)w'(u, v) = w(u, v) + h(u) - h(v).
4. Remove vertex and its edges.
5. Apply Dijkstra's algorithm for each vertex to find the shortest paths in the reweighted graph.
6. Convert the reweighted distances back to the original weights.

Pseudocode

```
function JohnsonAlgorithm(Graph):

  Add a new vertex q to Graph
  for each vertex v in Graph:
   Add edge from q to v with weight 0

  // Step 1: Run Bellman-Ford from vertex q
  if not BellmanFord(Graph, q):
    return "Graph contains a negative-weight cycle"

  // Step 2: Compute reweighting values
  for each vertex v in Graph:
    h(v) = distance from q to v

  // Step 3: Reweight the edges
  for each edge (u, v) in Graph:
    w'(u, v) = w(u, v) + h(u) - h(v)

  Remove vertex q and its edges from Graph

  // Step 4: Run Dijkstra for each vertex
  for each vertex u in Graph:
    distances[u] = Dijkstra(Graph, u, w')

  // Step 5: Adjust distances back to original weights
  for each pair of vertices (u, v):
    shortestPaths[u][v] = distances[u][v] - h(u) + h(v)

  return shortestPaths

function BellmanFord(Graph, source):
  Initialize distances[] and predecessors[]
  for i from 1 to |V| - 1:
    for each edge (u, v) in Graph:
      Relax the edge (u, v)
  for each edge (u, v) in Graph:
    if Relaxation is still possible:
      return false
  return true

function Dijkstra(Graph, source, weights):
  Initialize distances[] and priority queue
  while priority queue is not empty:
    Extract vertex with minimum distance
    for each neighbor v of u:
      Relax the edge (u, v)
  return distances
```

Implementation

Code in C++:

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <tuple>

using namespace std;

const int INF = numeric_limits<int>::max();

typedef pair<int, int> Edge; // Pair of destination and weight

typedef vector<vector<Edge>> Graph;

// Bellman-Ford Algorithm
bool bellmanFord(const Graph &graph, vector<int> &dist, int source) {
    int V = graph.size();
    dist.assign(V, INF);
    dist[source] = 0;

    for (int i = 0; i < V - 1; ++i) {
        for (int u = 0; u < V; ++u) {
            for (const auto &[v, weight] : graph[u]) {
                if (dist[u] != INF && dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                }
            }
        }
    }

    for (int u = 0; u < V; ++u) {
        for (const auto &[v, weight] : graph[u]) {
            if (dist[u] != INF && dist[u] + weight < dist[v]) {
                return false; // Negative-weight cycle detected
            }
```

```
        }
    }

    return true;
}

// Dijkstra's Algorithm
vector<int> dijkstra(const Graph &graph, int source) {
    int V = graph.size();
    vector<int> dist(V, INF);
    priority_queue<Edge, vector<Edge>, greater<Edge>> pq;

    dist[source] = 0;
    pq.emplace(0, source);

    while (!pq.empty()) {
        auto [currentDist, u] = pq.top();
        pq.pop();

        if (currentDist > dist[u]) continue;

        for (const auto &[v, weight] : graph[u]) {
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.emplace(dist[v], v);
            }
        }
    }

    return dist;
}

// Johnson's Algorithm
vector<vector<int>> johnson(const Graph &graph) {
    int V = graph.size();
    Graph modifiedGraph = graph;

    // Add a new vertex q
    modifiedGraph.emplace_back();
    for (int u = 0; u < V; ++u) {
        modifiedGraph.back().emplace_back(u, 0);
    }

    // Step 1: Run Bellman-Ford
    vector<int> h;
    if (!bellmanFord(modifiedGraph, h, V)) {
        throw runtime_error("Graph contains a negative-weight cycle");
    }

    // Step 2: Reweight edges
    Graph reweightedGraph(V);
    for (int u = 0; u < V; ++u) {
        for (const auto &[v, weight] : graph[u]) {
```

```
            reweightedGraph[u].emplace_back(v, weight + h[u] - h[v]);
        }
    }

    // Step 3: Run Dijkstra for each vertex
    vector<vector<int>> shortestPaths(V, vector<int>(V, INF));
    for (int u = 0; u < V; ++u) {
        vector<int> dist = dijkstra(reweightedGraph, u);
        for (int v = 0; v < V; ++v) {
            shortestPaths[u][v] = dist[v] == INF ? INF : dist[v] - h[u] + h[v];
        }
    }

    return shortestPaths;
}

int main() {
    // Example graph

    Graph graph = {
        {{1, 3}, {2, 8}},
        {{2, 2}, {3, 5}},
        {{3, 1}},
        {}
    };

    try {
        vector<vector<int>> result = johnson(graph);
        for (int i = 0; i < result.size(); ++i) {
            for (int j = 0; j < result[i].size(); ++j) {
                if (result[i][j] == INF) {
                    cout << "INF ";
                } else {
                    cout << result[i][j] << " ";
                }
            }
            cout << endl;
        }
    } catch (const exception &e) {
        cerr << e.what() << endl;
    }

    return 0;
}
```

Conclusion

Johnson’s Algorithm efficiently computes shortest paths in sparse graphs, even in the presence of negative weights. This lab implementation provided hands-on experience with algorithm design and highlighted the importance of edge reweighting to handle negative cycles.