Single Source Shortest Path

Ajeeta Asthana

April 2025

1 Dijkstra Algorithm

In this section, we discuss the Dijkstra algorithm. The objective of this algorithm is to find the shortest path from the source node to all other nodes in a weighted graph. The algorithm uses priority queue to greedily select the closest vertex that has not yet been processed, and performs the relaxation process on all of its outgoing edges.

1.1 Pseudo Code Implementation:

```
Dijkstra(G, W, s)   // uses priority queue
    Initialize (G, s)
    S <-- Null
    Q <-- V[G]
    while Q != null
        do u <-- ExtractMin(Q)  // Delete u from Q
            S = S U {u}
            for each vertex v E Adj[u]
                do RELAX(u, v, w) <-- this is an implicit DECREASE_KEY operation
```

[VK: Can you elaborate on the notations?. Add a line at the top of the pseudo-code to define the symbols.]

1.2 Data Structures Used:

```
typedef pair<int, int> pii;
typedef vector<vector<pii>> Graph;
```

- The graph is represented as an adjacency list where graph[u] contains all neighbors of vertex u
- Each neighbor is stored as a pair weight, vertex for efficient priority queue operations.

1.3 Main Algorithm Steps:

1. Initialization

```
vector<int> dist(n, numeric_limits<int>::max());
dist[source] = 0;
priority_queue<pii, vector<pii>, greater<pii>> pq;
pq.push({0, source});
```

- We set all distances to infinity (max int value) except the source (which is 0)
- A min-priority queue is created and the source is added with distance 0

1.4 Main Processing Loop:

```
while(!pq.empty()) {
    int u = pq.top().second;
    int dist_u = pq.top().first;

    pq.pop();

    if(dist_u > dist[u] {
        continue;
    }
    // Process Neighbors
}
```

- We repeatedly extract the vertex with the smallest distance from the priority queue.
- The distu ¿ dist[u] check helps avoid processing outdated entries in the priority queue.

1.5 Edge Relaxation:

```
for(const auto& edge : graph[u]) {
    int v = edge.second;    // Adjacent vertex
    int weight = edge.first;    // edge weight

    if(dist[u] != numeric_limit<int>::max() &&
        dist[u] + weight < dist[v]) {
            dist[v] = dist[u] + weight;
            pq.push({dist[v], v});
        }
}
```

- For each neighbor of the current vertex, we check if we can improve its distance.

- If the path through the current vertex u is shorter than the previously known shortest path to v, we update dist[v].
- We then add this vertex to the priority queue with its new distance.

1.6 Time and Space Complexity:

Time Complexity: O((V + E)log V) where V is the number of vertices and E is the number of edges

- Each vertex is extracted from the queue once: O(VlogV)
- Each edge is examined once: O(ElogV)

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

using namespace std;

// Typedefs for convenience
typedef pair<int, int> pii; (weight, vertex)
typedef vector<vector<pii>> Graph;

//Function to implement Dijkstra algorithm
vector<int> dijkstra(const Graph& graph, int source) {
    int n = graph.size();   // number of vertices

    // Distance array to store shortest distance from source to each vertex
    vector<int> dist(n, numeric_limits<int>::max());

    // Priority Queue to get the vertex with minimum distance
    // We use min-heap with custom comparison
    priority_queue<pii, vector<pii>, greater<pii>> pq;

    // Distance of source from itself to 0
    dist[source] = 0;
    pq.push({0, source}); // push (distance, vector)

    // Process vertices
    while(!pq.empty()) {
        // Get vertex with minimum distance
        int u = pq.top().second;
        int dist_u = pq.top().first;
        pq.pop();
```

```
        // If the popped vertex distance is greater than the calculated distance, skip
        if(dist_u > dist[u]) {
            continue;
        }

        // Check all adjacent vertices of u
        for(const auto& edge: graph[u]) {
            int v = edge.second;
            int weight = edge.first;

            // If there is a shorter path to v through u
            if(dist[u] != numeric_limits<int>::max() &&
                dist[u] + weight < dist[v] ) {
                    dist[v] = dist[u] + weight;
                    pq.push({dist[v], v});
                }
        }
    }
    return dist;
}
```

1.7 Example Execution:

In the example graph provided:

- We start at vertex 0 with distance 0
- Process neighbors at 0: update dist[1] = 4 and dist[2] = 3
- Next process vertex 2 (distance 3): update dist[3] = 7, dist[4] = 6
- Continue until all reachable vertices have their shortest paths calculated.

2 Bellman Ford Algorithm

The Bellman-Ford algorithm is a way to find single-source shortest paths in a graph with negative edge weights (but no negative cycles). The second loop of this algorithm also detects negative cycles.

The first loop relaxes each of the edges in the graph n-1 times. We claim that after n-1 iterations, the distances are guaranteed to be correct.

In general, the algorithm takes O(mn) time.

```
d[s] <-- 0
pii[s] <-- s
for each v E V - {s}
    do d[v] <-- inf
```

```
        pii[v] <-- nil

    for i <-- i to |V|-1
        do for each edge (u,v) E E
            do if d[v] > d[u] + w(u,v)
                pii[v] <-- u

    for each edge (u,v) E E
        do if d[v] > d[u] + w(u, v)
            then report negative cycle
```

A complete information of the Bellman-Ford Algorithm

```
#include <iostream>
#include <vector>
#include <limits>
#include <iomanip>
#include <fstream>
#include <sstream>

using namespace std;

// Structure to represent a weighted edge in the graph
struct Edge {
    int src, dest, weight;
};

// Function to implement Bellman-Ford algorithm
bool bellmanFord(const vector<Edge>& edges, int V, int source, vector<int>& dist, vector<int
    // Initialize distances from source to all vertices as infinite
    dist.assign(V, numeric_limits<int>::max());
    predecessor.assign(V, -1);

    // Distance of source vertex from itself is 0
    dist[source] = 0;

    // Relax all edges V-1 times
    for (int i = 1; i <= V - 1; i++) {
        bool anyUpdate = false;
        for (const auto& edge : edges) {
            int u = edge.src;
            int v = edge.dest;
            int weight = edge.weight;

            // If vertex u is reachable and we can get a shorter path to v
            if (dist[u] != numeric_limits<int>::max() && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
```

```
                predecessor[v] = u;
                anyUpdate = true;
            }
        }

        // If no update was made in this iteration, we can break early
        if (!anyUpdate) {
            break;
        }
    }

    // Check for negative-weight cycles
    for (const auto& edge : edges) {
        int u = edge.src;
        int v = edge.dest;
        int weight = edge.weight;

        if (dist[u] != numeric_limits<int>::max() && dist[u] + weight < dist[v]) {
            cout << "Graph contains negative weight cycle!" << endl;
            return false;  // Negative cycle exists
        }
    }

    return true;  // No negative cycle
}

// Function to reconstruct the shortest path from source to destination
vector<int> getPath(int dest, const vector<int>& predecessor) {
    vector<int> path;
    for (int at = dest; at != -1; at = predecessor[at]) {
        path.push_back(at);
    }
    reverse(path.begin(), path.end());
    return path;
}

// Function to print the paths
void printPaths(int source, const vector<int>& dist, const vector<int>& predecessor) {
    int V = dist.size();
    cout << "Shortest paths from vertex " << source << ":\n";

    for (int i = 0; i < V; i++) {
        if (i != source) {
            cout << "To vertex " << i << " (distance: ";
            if (dist[i] == numeric_limits<int>::max()) {
                cout << "INF): Not reachable" << endl;
```

```
            } else {
                cout << dist[i] << "): ";
                vector<int> path = getPath(i, predecessor);
                for (size_t j = 0; j < path.size(); j++) {
                    cout << path[j];
                    if (j < path.size() - 1) cout << " -> ";
                }
                cout << endl;
            }
        }
    }
}

// Function to read graph from file (optional)
bool readGraphFromFile(const string& filename, vector<Edge>& edges, int& V) {
    ifstream file(filename);
    if (!file.is_open()) {
        cout << "Could not open file: " << filename << endl;
        return false;
    }

    file >> V;
    int E;
    file >> E;

    edges.resize(E);
    for (int i = 0; i < E; i++) {
        file >> edges[i].src >> edges[i].dest >> edges[i].weight;
    }

    file.close();
    return true;
}

// Function to create a graph manually
void createGraphManually(vector<Edge>& edges, int& V) {
    cout << "Enter number of vertices: ";
    cin >> V;

    int E;
    cout << "Enter number of edges: ";
    cin >> E;

    cout << "Enter edge information (source destination weight):" << endl;
    edges.resize(E);
    for (int i = 0; i < E; i++) {
```

```
        cout << "Edge " << i+1 << ": ";
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight;
    }
}

int main() {
    vector<Edge> edges;
    int V;
    int choice;

    cout << "=== Bellman-Ford Shortest Path Algorithm ===" << endl;
    cout << "1. Use example graph" << endl;
    cout << "2. Create graph manually" << endl;
    cout << "3. Read graph from file (format: V E followed by E lines of source dest weight)
    cout << "Enter choice: ";
    cin >> choice;

    switch (choice) {
        case 1: {
            // Example graph
            V = 5;
            edges = {
                {0, 1, -1}, {0, 2, 4}, {1, 2, 3}, {1, 3, 2},
                {1, 4, 2}, {3, 2, 5}, {3, 1, 1}, {4, 3, -3}
            };
            break;
        }
        case 2:
            createGraphManually(edges, V);
            break;
        case 3: {
            string filename;
            cout << "Enter filename: ";
            cin >> filename;
            if (!readGraphFromFile(filename, edges, V)) {
                return 1;
            }
            break;
        }
        default:
            cout << "Invalid choice!" << endl;
            return 1;
    }

    int source;
    cout << "Enter source vertex (0 to " << V-1 << "): ";
```

```
    cin >> source;

    // Validate source vertex
    if (source < 0 || source >= V) {
        cout << "Invalid source vertex!" << endl;
        return 1;
    }

    vector<int> dist;
    vector<int> predecessor;

    // Run Bellman-Ford algorithm
    if (bellmanFord(edges, V, source, dist, predecessor)) {
        printPaths(source, dist, predecessor);
    } else {
        cout << "Cannot find shortest paths due to negative cycle" << endl;
    }

    return 0;
}
```