Dijkstra

Given a directed or undirected weighted graph with V vertices and E edges with no negative cycles, the problem is to find the lengths of the shortest paths from a starting vertex s to all other vertices, and find the shortest paths themselves. This problem is called *single-source shortest paths problem*.

We maintain an array $d[]$ where for every vertex u, $d[u]$ is the length of the shortest path from s to u. We initialize $d[u] = \infty \; \forall \; u \in V - \{s\}$ where ∞ is some large number. Also, $d[s] = 0$.

We also maintain an array $p[]$ which keeps track of immediate predecessor, i.e., $p[v]$ would store the vertex which is predecessor of v. This array would be helpful in restoring the path from source vertex s to any other vertex t.

The algorithm runs for almost V iterations.
At each iteration, an unmarked vertex u with least value $d[u]$ is extracted and marked.
From vertex u, relaxations are performed on all vertices adjacent to it. That is, for each adjacent vertex v, we minimize the value of $d[v]$. If the current edge length is len, then

$$d[v] = min(d[v], len + d[u])$$

On every relaxation we perform, we update the value of $p[v]$ as:

$$p[v] = u$$

The time for extracting an unmarked vertex u with least value $d[u]$ is naively $O(V)$ if we just loop through all vertices. The step of extraction is performed V times for every vertex. Since the time for relaxation of an edge is simply $O(1)$ and we perform the step of relaxation E times for every edge. The time complexity of this algorithm would be $O(V^2 + E)$.

We can use data structures such as AVL trees, binary heap, etc. to solve this problem of extracting minimum in $O(\log V)$. For example, we can make a compromise and use a binary heap for both operations of extraction (in $O(\log V)$) and updating the value after relaxation (in $O(\log V)$).
Then, the total complexity would be,
$O(V \log V + E \log V) = O((V + E) \log V) \approx O(E \log V)$.

Here is an implementation for the algorithm using the `graph.hpp` header introduced before. The file `dijkstra.hpp` has `struct Dijkstra` inherited publicly from `class weightedGraph` in `graph.hpp`.

```
// dijkstra.hpp

#ifndef DIJKSTRA_HPP
#define DIJKSTRA_HPP

#include "graph.hpp"

struct Dijkstra : public weightedGraph {
    using ll = long long int;
    const ll INF = 0x3f3f3f3f3f3f3f3f;

    ll *d;
    Vertex *p;
```

```
    /**
     * @brief Constructs a new Dijkstra object.
     *
     * @param _V Number of vertices.
     * @param _directed Default = false. specify true if graph is directed.
     */
    Dijkstra(Vertex _V, bool _directed = false) : weightedGraph(_V, _directed) {
        d = new ll[V + 1]();
        p = new Vertex[V + 1]();
    }

    /**
     * @brief Constructs a new Dijkstra object from a weighted Graph object.
     *
     * @param G The weighted Graph Object to be copied.
     */
    Dijkstra(weightedGraph G) : weightedGraph(G) {
        d = new ll[V + 1];
        p = new Vertex[V + 1]();
    }

    /**
     * @brief Constructs a new Dijkstra object from an old one.
     * It calls the weightedGraph copy constructor.
     *
     * @param copy The old Dijkstra object to be copied.
     */
    Dijkstra(const Dijkstra &copy) : weightedGraph(copy) {
        memcpy(d, copy.d, sizeof(ll) * (V + 1));
    }


    /**
     * @brief Destroys the Dijkstra object.
     *
     */
    ~Dijkstra() {
        delete []d;
    }

    /**
     * @brief finds the lengths of shortest paths from the source vertex to all
     * vertices and stores in Dijkstra::d[].
     *
     * @param s Default = 1. The source vertex.
     */
    void solveShortestPaths(Vertex s = 1) {
        using length = std::pair<ll, Vertex>;
        std::priority_queue<length, std::vector<length>, std::greater<length>>
                                                                              pq;
        memset(d, 0x3f, sizeof(ll) * (V + 1)); // initializing d[] to INF
        d[s] = 0LL;
        p[s] = -1;
        pq.push(length(d[s], s));
        while (not pq.empty()) {
```

```cpp
            Vertex u = pq.top().second;
            ll dist = pq.top().first;
            pq.pop();

            if (dist > d[u])
                continue;

            for (Edge e : adj[u]) {
                auto[v, len] = e;
                if (len + d[u] < d[v]) {
                    d[v] = len + d[u];
                    p[v] = u;
                    pq.push(length(d[v], v));
                }
            }
        }
    }

    /**
     * @brief prints a path from start vertex s to destination vertex t. Also
     * returns a vector container having the path.
     *
     * @param s Start vertex s.
     * @param t Destination vertex t.
     * @return std::vector<Vertex> Vector Container having the correct order
     * of vertices in its path.
     */
    std::vector<Vertex> printPath(Vertex s, Vertex t) {
        std::vector<Vertex> path;
        path.reserve(V);
        for (Vertex u = t; u != s; u = p[u])
            path.push_back(u);
        path.push_back(s);
        std::reverse(path.begin(), path.end());
        for (int u : path) {
            std::cout << u << ' ';
        }
        std::cout << "\n";
        return path;
    }

};
```

The C++ STL `std::priority_queue` uses a binary heap. Therefore, the push and pop operations would be $O(\log V)$.
Hence, the above function `solveShortestPaths(s)` runs in $O(E \log V)$.

The process of printing is also very simple. The idea is to store the parent of destination vertex u, $p[u]$ in a container and assign u as $p[u]$. The function `printPath(s, t)` runs in $O(V)$.