Lecture 11: Minimum Spanning Trees and Shortest Paths

C++ Code Samples — Sedgwick Algorithms Course — lecture-11-samples.cpp

```cpp
/******************************************************************************
 * Lecture 11: Minimum Spanning Trees and Shortest Paths
 *
 * Topics covered:
 *   1. Weighted edge and weighted graph structures
 *   2. Kruskal's MST with Union-Find
 *   3. Prim's MST (eager, using min-heap)
 *   4. Dijkstra's shortest path algorithm
 *   5. Demo: show MST edges and total weight, shortest distances
 *
 * Compile: g++ -std=c++17 -o lecture-11 lecture-11-samples.cpp
 * Run:     ./lecture-11
 ******************************************************************************/

#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
#include <numeric>

using namespace std;

// === SECTION: Weighted Edge and Graph Structures ===

struct Edge {
    int u, v;
    double weight;

    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

class WeightedGraph {
public:
    int V;
    vector<vector<pair<int, double>>> adj;  // adj[u] = {(v, weight), ...}
    vector<Edge> edges;                      // all edges (for Kruskal)

    WeightedGraph(int V) : V(V), adj(V) {}

    void addEdge(int u, int v, double w) {
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
        edges.push_back({u, v, w});
    }
};

// === SECTION: Union-Find (Disjoint Set) ===
// Used by Kruskal's algorithm to efficiently detect cycles.
// With path compression and union by rank: nearly O(1) per operation.

class UnionFind {
public:
    vector<int> parent, rank_;
    int components;

    UnionFind(int n) : parent(n), rank_(n, 0), components(n) {
        iota(parent.begin(), parent.end(), 0);  // parent[i] = i
    }
```

```cpp
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // path compression
        }
        return parent[x];
    }

    bool unite(int x, int y) {
        int rx = find(x), ry = find(y);
        if (rx == ry) return false;  // already in same set

        // Union by rank: attach smaller tree under larger tree
        if (rank_[rx] < rank_[ry]) swap(rx, ry);
        parent[ry] = rx;
        if (rank_[rx] == rank_[ry]) rank_[rx]++;
        components--;
        return true;
    }

    bool connected(int x, int y) {
        return find(x) == find(y);
    }
};

// === SECTION: Kruskal's MST ===
// Greedy algorithm: sort all edges by weight, then add edges that don't
// create a cycle (checked via Union-Find).
// Time: O(E log E) for sorting.

struct MSTResult {
    vector<Edge> mstEdges;
    double totalWeight;
};

MSTResult kruskalMST(const WeightedGraph& g) {
    MSTResult result;
    result.totalWeight = 0;

    // Sort edges by weight
    vector<Edge> sortedEdges = g.edges;
    sort(sortedEdges.begin(), sortedEdges.end());

    UnionFind uf(g.V);

    for (const Edge& e : sortedEdges) {
        // If u and v are in different components, adding this edge is safe
        if (uf.unite(e.u, e.v)) {
            result.mstEdges.push_back(e);
            result.totalWeight += e.weight;

            // MST has exactly V-1 edges
            if ((int)result.mstEdges.size() == g.V - 1) break;
        }
    }
    return result;
}

// === SECTION: Prim's MST (Eager, Min-Heap) ===
// Start from vertex 0. Maintain a priority queue of edges crossing the cut
// between the MST and the rest of the graph. Always pick the lightest edge.
// Time: O(E log V) with a binary heap.

MSTResult primMST(const WeightedGraph& g) {
```

```cpp
    MSTResult result;
    result.totalWeight = 0;

    // minDist[v] = minimum edge weight connecting v to the current MST
    vector<double> minDist(g.V, 1e18);
    vector<int> parent(g.V, -1);
    vector<bool> inMST(g.V, false);

    // Min-heap: (weight, vertex)
    priority_queue<pair<double, int>, vector<pair<double, int>>,
                   greater<pair<double, int>>> pq;

    minDist[0] = 0;
    pq.push({0, 0});

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

        if (inMST[u]) continue;
        inMST[u] = true;
        result.totalWeight += dist;

        if (parent[u] != -1) {
            result.mstEdges.push_back({parent[u], u, dist});
        }

        // Relax edges from u
        for (auto [v, w] : g.adj[u]) {
            if (!inMST[v] && w < minDist[v]) {
                minDist[v] = w;
                parent[v] = u;
                pq.push({w, v});
            }
        }
    }
    return result;
}

// === SECTION: Dijkstra's Shortest Path ===
// Find shortest paths from a single source to all other vertices in a
// non-negative weighted graph.
// Greedy: always process the vertex with the smallest known distance.
// Time: O(E log V) with a binary heap.

struct DijkstraResult {
    vector<double> dist;
    vector<int> parent;
};

DijkstraResult dijkstra(const WeightedGraph& g, int source) {
    DijkstraResult result;
    result.dist.assign(g.V, 1e18);
    result.parent.assign(g.V, -1);

    // Min-heap: (distance, vertex)
    priority_queue<pair<double, int>, vector<pair<double, int>>,
                   greater<pair<double, int>>> pq;

    result.dist[source] = 0;
    pq.push({0, source});

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

```cpp
        pq.pop();

        // Skip if we already found a shorter path to u
        if (d > result.dist[u]) continue;

        // Relax each neighbor
        for (auto [v, w] : g.adj[u]) {
            double newDist = result.dist[u] + w;
            if (newDist < result.dist[v]) {
                result.dist[v] = newDist;
                result.parent[v] = u;
                pq.push({newDist, v});
            }
        }
    }
    return result;
}

// Reconstruct shortest path from source to target
vector<int> dijkstraPath(const DijkstraResult& res, int target) {
    vector<int> path;
    if (res.dist[target] >= 1e18) return path;
    for (int v = target; v != -1; v = res.parent[v]) {
        path.push_back(v);
    }
    reverse(path.begin(), path.end());
    return path;
}

// === MAIN: Demos ===

int main() {
    cout << "==================================================\n";
    cout << "  Lecture 11: MST and Shortest Paths\n";
    cout << "==================================================\n\n";

    // Sample weighted graph:
    //       1
    //   0 ----- 1
    //   |  \    |  \
    //  4|   2\  |3   6\
    //   |     \ |      \
    //   3 ----- 2 ----- 4
    //       5       7
    //
    // Edges: (0,1,1) (0,2,2) (0,3,4) (1,2,3) (1,4,6) (2,3,5) (2,4,7)

    WeightedGraph g(5);
    g.addEdge(0, 1, 1);
    g.addEdge(0, 2, 2);
    g.addEdge(0, 3, 4);
    g.addEdge(1, 2, 3);
    g.addEdge(1, 4, 6);
    g.addEdge(2, 3, 5);
    g.addEdge(2, 4, 7);

    // --- Kruskal's MST ---
    cout << "--- Kruskal's MST ---\n";
    MSTResult kruskal = kruskalMST(g);
    cout << "  MST edges:\n";
    for (const Edge& e : kruskal.mstEdges) {
        cout << "    " << e.u << " -- " << e.v
             << "  (weight " << e.weight << ")\n";
    }
```

```cpp
    cout << "  Total MST weight: " << kruskal.totalWeight << "\n\n";

    // --- Prim's MST ---
    cout << "--- Prim's MST ---\n";
    MSTResult prim = primMST(g);
    cout << "  MST edges:\n";
    for (const Edge& e : prim.mstEdges) {
        cout << "    " << e.u << " -- " << e.v
             << "  (weight " << e.weight << ")\n";
    }
    cout << "  Total MST weight: " << prim.totalWeight << "\n\n";

    // --- Dijkstra's Shortest Path ---
    cout << "--- Dijkstra's Shortest Path (source=0) ---\n";
    DijkstraResult dijk = dijkstra(g, 0);
    cout << "  Shortest distances from vertex 0:\n";
    for (int i = 0; i < g.V; i++) {
        cout << "    0 -> " << i << " : " << dijk.dist[i];
        vector<int> path = dijkstraPath(dijk, i);
        cout << "  path: ";
        for (int j = 0; j < (int)path.size(); j++) {
            if (j > 0) cout << " -> ";
            cout << path[j];
        }
        cout << "\n";
    }
    cout << "\n";

    // --- Comparison ---
    cout << "--- Key Differences ---\n";
    cout << "  Kruskal: sorts all edges globally, uses Union-Find\n";
    cout << "           Best for sparse graphs (E close to V)\n";
    cout << "  Prim:    grows MST from a starting vertex, uses min-heap\n";
    cout << "           Best for dense graphs (E close to V^2)\n";
    cout << "  Both produce a MST with the same total weight: "
         << kruskal.totalWeight << "\n";

    return 0;
}
```