Lecture 10: Breadth-First Search and Graph Applications

C++ Code Samples — Sedgwick Algorithms Course — lecture-10-samples.cpp

```cpp
/**************************************************************************
 * Lecture 10: Breadth-First Search and Graph Applications
 *
 * Topics covered:
 *   1. BFS traversal
 *   2. Shortest path in unweighted graph
 *   3. Bipartite checking (two-coloring)
 *   4. Topological sort (Kahn's algorithm with in-degrees)
 *   5. Demo: find shortest path between two nodes
 *
 * Compile: g++ -std=c++17 -o lecture-10 lecture-10-samples.cpp
 * Run:     ./lecture-10
 **************************************************************************/

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

using namespace std;

// Undirected graph for BFS demos
class Graph {
public:
    int V;
    vector<vector<int>> adj;

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

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

// Directed graph for topological sort
class Digraph {
public:
    int V;
    vector<vector<int>> adj;

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

    void addEdge(int u, int v) {
        adj[u].push_back(v);    // directed: u -> v
    }
};

// === SECTION: BFS Traversal ===
// Explore vertices in layers: first the start, then all neighbors at distance 1,
// then distance 2, etc. Uses a FIFO queue. Time: O(V + E).

vector<int> bfsTraversal(const Graph& g, int start) {
    vector<bool> visited(g.V, false);
    vector<int> order;
    queue<int> q;

    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
```

```
        int u = q.front();
        q.pop();
        order.push_back(u);

        for (int v : g.adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
    return order;
}

// === SECTION: Shortest Path in Unweighted Graph ===
// BFS naturally finds shortest paths (fewest edges) from a source.
// We store the distance and predecessor for each vertex.

struct BFSResult {
    vector<int> dist;   // dist[v] = number of edges from source to v
    vector<int> parent; // parent[v] = predecessor on shortest path
};

BFSResult bfsShortestPath(const Graph& g, int source) {
    BFSResult result;
    result.dist.assign(g.V, -1);
    result.parent.assign(g.V, -1);

    queue<int> q;
    result.dist[source] = 0;
    q.push(source);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v : g.adj[u]) {
            if (result.dist[v] == -1) {  // not yet visited
                result.dist[v] = result.dist[u] + 1;
                result.parent[v] = u;
                q.push(v);
            }
        }
    }
    return result;
}

// Reconstruct the path from source to target using the parent array.
vector<int> reconstructPath(const BFSResult& bfs, int target) {
    vector<int> path;
    if (bfs.dist[target] == -1) return path;  // unreachable

    for (int v = target; v != -1; v = bfs.parent[v]) {
        path.push_back(v);
    }
    reverse(path.begin(), path.end());
    return path;
}

// === SECTION: Bipartite Checking (Two-Coloring) ===
// A graph is bipartite iff it can be two-colored: assign each vertex color 0 or 1
// such that no two adjacent vertices share the same color.
// Equivalent to: the graph has no odd-length cycles.
// Approach: BFS and try to assign alternating colors.
```

```
bool isBipartite(const Graph& g) {
    vector<int> color(g.V, -1);  // -1 = uncolored

    // Check each component (graph may be disconnected)
    for (int start = 0; start < g.V; start++) {
        if (color[start] != -1) continue;

        queue<int> q;
        color[start] = 0;
        q.push(start);

        while (!q.empty()) {
            int u = q.front();
            q.pop();

            for (int v : g.adj[u]) {
                if (color[v] == -1) {
                    color[v] = 1 - color[u];  // assign opposite color
                    q.push(v);
                } else if (color[v] == color[u]) {
                    return false;  // same color on adjacent vertices
                }
            }
        }
    }
    return true;
}

// === SECTION: Topological Sort (Kahn's Algorithm) ===
// For a DAG (directed acyclic graph), produce a linear ordering of vertices
// such that for every directed edge u->v, u appears before v.
//
// Kahn's algorithm:
//   1. Compute in-degree of each vertex.
//   2. Enqueue all vertices with in-degree 0.
//   3. Repeatedly dequeue a vertex, add to result, and decrement
//      in-degrees of its neighbors. Enqueue neighbors that reach in-degree 0.
//   4. If result has fewer than V vertices, the graph has a cycle.

vector<int> topologicalSort(const Digraph& g) {
    vector<int> inDegree(g.V, 0);

    // Step 1: compute in-degrees
    for (int u = 0; u < g.V; u++)
        for (int v : g.adj[u])
            inDegree[v]++;

    // Step 2: enqueue vertices with in-degree 0
    queue<int> q;
    for (int v = 0; v < g.V; v++)
        if (inDegree[v] == 0) q.push(v);

    // Step 3: process
    vector<int> order;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        order.push_back(u);
```

```
        for (int v : g.adj[v]) {
            indegree[v]--;
            if (indegree[v] == 0) {
                q.push(v);
            }
        }
    }

    // Step 4: check for cycle
    if (int(order.size()) != n) {
        cout << "  WARNING: Graph has a cycle, topological sort not possible.\n";
    }
    return order;
}

// === MAIN: Demos ===

int main() {
    cout << "==================================================\n";
    cout << "  Lecture 10: BFS and Graph Applications\n";
    cout << "==================================================\n\n";

    // Sample undirected graph:
    //   0 --- 1 --- 2
    //   |           |
    //   3 --- 4 --- 5
    //         |
    //         6
    Graph g(7);
    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(0, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 5);
    g.addEdge(2, 5);
    g.addEdge(4, 6);

    // --- BFS Traversal ---
    cout << "--- BFS Traversal (start=0) ---\n";
    vector<int> bfsOrder = g.bfsTraversal(0);
    cout << "  Visit order: ";
    for (int v : bfsOrder) cout << v << " ";
    cout << "\n\n";

    // --- Shortest Path ---
    cout << "--- Shortest Path (source=0) ---\n";
    BFSResult res = g.bfsShortestPaths(0);
    cout << "  Distances from vertex 0:\n";
    for (int i = 0; i < g.V; ++i) {
        cout << "    0 -> " << i << " : " << res.dist[i] << " edges\n";
    }
    cout << "\n";

    // Demo: path from 0 to 6
    cout << "--- Path from 0 to 6 ---\n";
    vector<int> path = reconstructPath(res, 6);
    cout << "  Path: ";
    for (int i = 0; i < int(path.size()); ++i) {
        if (i > 0) cout << " -> ";
        cout << path[i];
    }
    cout << " (length " << res.dist[6] << ")\n\n";
```

```
    // --- Bipartite Check ---
    cout << "--- Bipartite Checking ---\n"

    // Graph g has cycle 0-1-2-5-4-3-0 (length 6, even) => bipartite
    cout << "  Graph g: " << (isBipartite(g) ? "bipartite" : "NOT bipartite")
         << "\n"

    // Add edge to create odd cycle: 0-1, 1-2, 2-0 (triangle)
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 0);    // triangle => odd cycle => not bipartite
    g2.addEdge(2, 3);
    cout << "  Triangle graph (0-1-2-0): "
         << (isBipartite(g2) ? "bipartite" : "NOT bipartite") << "\n\n"

    // --- Topological Sort ---
    cout << "--- Topological Sort (Kahn's Algorithm) ---\n"
    // DAG representing course prerequisites:
    //   0: Intro to CS
    //   1: Data Structures (requires 0)
    //   2: Algorithms (requires 1)
    //   3: Discrete Math
    //   4: Graph Theory (requires 1, 3)
    //   5: Advanced Algorithms (requires 2, 4)
    Digraph dag(6);
    dag.addEdge(0, 1);    // Intro -> DS
    dag.addEdge(1, 2);    // DS -> Algorithms
    dag.addEdge(1, 4);    // DS -> Graph Theory
    dag.addEdge(3, 4);    // Discrete Math -> Graph Theory
    dag.addEdge(2, 5);    // Algorithms -> Advanced
    dag.addEdge(4, 5);    // Graph Theory -> Advanced

    vector<string> courseNames = {
        "Intro CS", "Data Structures", "Algorithms",
        "Discrete Math", "Graph Theory", "Advanced Algorithms"
    };

    vector<int> topoOrder = topologicalSort(dag);
    cout << "  Valid course order:\n"
    for (int i = 0; i < topoOrder.size(); ++i) {
        cout << "    " << (i + 1) << ". " << courseNames[topoOrder[i]]
             << " (vertex " << topoOrder[i] << ")\n"
    }

    return 0
}
```