Lecture 09: Graph Fundamentals and Depth-First Search

C++ Code Samples — Sedgwick Algorithms Course — lecture-09-samples.cpp

```cpp
/*****************************************************************************
 * Lecture 09: Graph Fundamentals and Depth-First Search
 *
 * Topics covered:
 *   1. Adjacency list graph representation
 *   2. Adjacency matrix graph representation
 *   3. DFS recursive traversal
 *   4. DFS iterative traversal (using stack)
 *   5. Cycle detection in undirected graph
 *   6. Connected components
 *
 * Compile: g++ -std=c++17 -o lecture-09 lecture-09-samples.cpp
 * Run:     ./lecture-09
 *****************************************************************************/

#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>

using namespace std;

// === SECTION: Adjacency List Representation ===
// Each vertex stores a list of its neighbors. Space: O(V + E).

class GraphAdjList {
public:
    int V;
    vector<vector<int>> adj;

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

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

    void print() const {
        for (int v = 0; v < V; ++v) {
            cout << v << ": ";
            for (int i = 0; i < (int)adj[v].size(); ++i) {
                if (i > 0) cout << ", ";
                cout << adj[v][i];
            }
            cout << "\n";
        }
    }
};

// === SECTION: Adjacency Matrix Representation ===
// A V x V matrix where mat[u][v] = 1 if edge (u,v) exists. Space: O(V^2).

class GraphAdjMatrix {
public:
    int V;
    vector<vector<int>> mat;

    GraphAdjMatrix(int V) : V(V), mat(V, vector<int>(V, 0)) {}

    void addEdge(int u, int v) {
        mat[u][v] = 1;
```

```
        matrix[v][u] = 1;  // undirected
    }

    void print() const {
        cout << "    ";
        for (int i = 0; i < V; ++i) cout << i << " ";
        cout << "\n";
        for (int i = 0; i < V; ++i) {
            cout << i << "  " << adj[i].size() << " ";
            for (int j = 0; j < V; ++j) {
                cout << matrix[i][j] << " ";
            }
            cout << "\n";
        }
    }
};

// === SECTION: DFS Recursive Traversal ===
// Visit a vertex, mark it, then recursively visit all unmarked neighbors.
// Time: O(V + E).

void dfsRecursiveHelper(const Graph& g, int v, vector<bool>& visited,
                        vector<int>& order) {
    visited[v] = true;
    order.push_back(v);
    for (int w : g.adj[v]) {
        if (!visited[w]) {
            dfsRecursiveHelper(g, w, visited, order);
        }
    }
}

vector<int> dfsRecursive(const Graph& g, int start) {
    vector<bool> visited(g.V, false);
    vector<int> order;
    dfsRecursiveHelper(g, start, visited, order);
    return order;
}

// === SECTION: DFS Iterative Traversal (Using Stack) ===
// Same traversal order concept as recursive DFS, but uses an explicit stack.
// Note: the exact visit order may differ from recursive DFS because neighbors
// are pushed in forward order and popped in reverse.

vector<int> dfsIterative(const Graph& g, int start) {
    vector<bool> visited(g.V, false);
    vector<int> order;
    stack<int> st;

    st.push(start);
    while (!st.empty()) {
        int v = st.top();
        st.pop();

        if (visited[v]) continue;
        visited[v] = true;
        order.push_back(v);

        // Push neighbors in reverse order so that the first neighbor
        // in the adjacency list is visited first.
        for (int i = (int)g.adj[v].size() - 1; i >= 0; --i) {
            int w = g.adj[v][i];
            if (!visited[w]) {
                st.push(w);
```

```
    return order;
}

// === SECTION: Cycle Detection in Undirected Graph ===
// During DFS, if we encounter a visited vertex that is NOT the parent
// of the current vertex, we have found a cycle.

bool hasCycleDFS(const vector<vector<int>>& g, int v, int parent,
                 vector<bool>& visited) {
    visited[v] = true;
    for (int w : g[v]) {
        if (!visited[w]) {
            if (hasCycleDFS(g, w, v, visited)) return true;
        } else if (w != parent) {
            // Found a back edge to a visited vertex that is not our parent
            return true;
        }
    }
    return false;
}

bool hasCycle(const vector<vector<int>>& g) {
    vector<bool> visited(g.size(), false);
    for (int v = 0; v < (int)g.size(); ++v) {
        if (!visited[v]) {
            if (hasCycleDFS(g, v, -1, visited)) return true;
        }
    }
    return false;
}

// === SECTION: Connected Components ===
// Run DFS from each unvisited vertex. Each DFS call discovers one component.
// Two vertices are connected iff they have the same component id.

vector<int> connectedComponents(const vector<vector<int>>& g) {
    vector<int> comp(g.size(), -1);
    int numComponents = 0;

    for (int v = 0; v < (int)g.size(); ++v) {
        if (comp[v] == -1) {
            // BFS/DFS to label all vertices in this component
            stack<int> st;
            st.push(v);
            while (!st.empty()) {
                int u = st.top(); st.pop();
                if (comp[u] != -1) continue;
                comp[u] = numComponents;
                for (int w : g[u]) {
                    if (comp[w] == -1) st.push(w);
                }
            }
            numComponents++;
        }
    }
    return comp;
}

// === MAIN: Demo with a Sample Graph ===
```

```cpp
int main() {
    cout << "================================================\n";
    cout << "  Lecture 09: Graph Fundamentals and DFS\n";
    cout << "================================================\n\n";

    // Sample graph:
    //   0 --- 1 --- 2
    //   |     |
    //   3     4 --- 5
    //               |
    //         6 --- 7    (separate component: 6-7)

    // --- Adjacency List ---
    cout << "--- Adjacency List Representation ---\n";
    Graph g(8);
    g.addEdge(0, 1);
    g.addEdge(0, 3);
    g.addEdge(1, 2);
    g.addEdge(1, 4);
    g.addEdge(4, 5);
    g.addEdge(6, 7);
    g.print();
    cout << "\n";

    // --- Adjacency Matrix ---
    cout << "--- Adjacency Matrix Representation ---\n";
    GraphMatrix gm(8);
    gm.addEdge(0, 1);
    gm.addEdge(0, 3);
    gm.addEdge(1, 2);
    gm.addEdge(1, 4);
    gm.addEdge(4, 5);
    gm.addEdge(6, 7);
    gm.print();
    cout << "\n";

    // --- DFS Recursive ---
    cout << "--- DFS Recursive (start=0) ---\n";
    vector<int> recOrder = g.dfsRecursive(0);
    cout << "  Visit order: ";
    for (int v : recOrder) cout << v << " ";
    cout << "\n\n";

    // --- DFS Iterative ---
    cout << "--- DFS Iterative (start=0) ---\n";
    vector<int> iterOrder = g.dfsIterative(0);
    cout << "  Visit order: ";
    for (int v : iterOrder) cout << v << " ";
    cout << "\n\n";

    // --- Cycle Detection ---
    cout << "--- Cycle Detection ---\n";
    cout << "  Graph without extra edge: "
         << (g.hasCycle() ? "HAS cycle" : "No cycle") << "\n";

    // Add edge 2-4 to create a cycle: 1-2, 2-4, 4-1
    Graph g2(8);
    g2.addEdge(0, 1); g2.addEdge(0, 3); g2.addEdge(1, 2);
    g2.addEdge(1, 4); g2.addEdge(4, 5); g2.addEdge(6, 7);
    g2.addEdge(2, 4);   // creates cycle 1-2-4-1
    cout << "  After adding edge 2-4: "
         << (g2.hasCycle() ? "HAS cycle" : "No cycle") << "\n\n";

    // --- Connected Components ---
```

```
    cout << "--- Connected Components ---\n"
    vector<int> comp = connectedComponents(g);
    int numComp = *max_element(comp.begin(), comp.end()) + 1
    cout << "  Number of components: " << numComp << "\n"
    for (int c = 0; c < numComp; ++c) {
        cout << "  Component " << c << ": "
        for (int v = 0; v < g.V; ++v) {
            if (comp[v] == c) cout << v << " "
        }
        cout << "\n"
    }

    return 0
}
```