```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <set>
#include <queue>
#include <stack>
#include <functional>
#include <string>

using namespace std;

class Graph {
private:
  unordered_map<int, vector<int>> adjacencyList;

public:
  Graph() = default;

  Graph(const unordered_map<int, vector<int>>& adjList) {
    adjacencyList = adjList;
  }

  void addVertex(int vertex) {
    if (adjacencyList.find(vertex) == adjacencyList.end()) {
      adjacencyList[vertex] = vector<int>();
    }
  }

  void addEdge(int vertex1, int vertex2) {
    addVertex(vertex1);
    addVertex(vertex2);
    adjacencyList[vertex1].push_back(vertex2);
    adjacencyList[vertex2].push_back(vertex1);
  }

  void printGraph() {
    cout << "\nGraph representation (Adjacency List):" << endl;
    for (const auto& pair : adjacencyList) {
      cout << pair.first << " -> ";
      for (int neighbor : pair.second) {
        cout << neighbor << " ";
```

```
    }
    cout << endl;
  }
}

void dfsRecursiveHelper(int vertex, set<int>& visited, vector<int>& result,int level) {
  visited.insert(vertex);
  result.push_back(vertex);
  cout<<"Node: "<<vertex<<" Level: "<<level<<endl;
  for (int neighbor : adjacencyList[vertex]) {
    if (visited.find(neighbor) == visited.end()) {
      dfsRecursiveHelper(neighbor, visited, result,level+1);
    }
  }
}

vector<int> dfsRecursive(int startVertex) {
  set<int> visited;
  vector<int> result;

  dfsRecursiveHelper(startVertex, visited, result,0);
  return result;
}

vector<int> dfsNonRecursive(int startVertex) {
  vector<int> result;
  set<int> visited;
  stack<int> stack;
  stack.push(startVertex);
  while (!stack.empty()) {
    int currentVertex = stack.top();
    stack.pop();
    if (visited.find(currentVertex) != visited.end()) continue;
    visited.insert(currentVertex);
    result.push_back(currentVertex);
    for (int i = adjacencyList[currentVertex].size() - 1; i >= 0; i--) {
      int neighbor = adjacencyList[currentVertex][i];
      if (visited.find(neighbor) == visited.end()) {
        stack.push(neighbor);
      }
    }
```

```
    }
    return result;
  }

  vector<int> bfs(int startVertex) {
    vector<int> result;
    set<int> visited;
    queue<pair<int,int>> queue;
    visited.insert(startVertex);
    queue.push({startVertex,0});
    while (!queue.empty()) {
      int currentVertex = queue.front().first;
      int level = queue.front().second;
      queue.pop();
      result.push_back(currentVertex);
      cout<<"Node: "<<currentVertex<<" Level: "<<level<<endl;
      for (int neighbor : adjacencyList[currentVertex]) {
        if (visited.find(neighbor) == visited.end()) {
          visited.insert(neighbor);
          queue.push({neighbor,level+1});
        }
      }
    }
    return result;
  }

  bool dlsRecursiveHelper(int vertex, int target, int depthLimit, set<int>& visited,
vector<int>& result) {
    if (vertex == target) {
      result.push_back(vertex);
      return true;
    }
    if (depthLimit <= 0) return false;
    visited.insert(vertex);
    result.push_back(vertex);
    for (int neighbor : adjacencyList[vertex]) {
      if (visited.find(neighbor) == visited.end()) {
        bool found = dlsRecursiveHelper(neighbor, target, depthLimit - 1, visited, result);
        if (found) return true;
      }
    }
```

```
        result.pop_back();
        visited.erase(vertex);
        return false;
    }

    vector<int> depthLimitedSearch(int startVertex, int target, int depthLimit) {
        set<int> visited;
        vector<int> result;
        bool found = dlsRecursiveHelper(startVertex, target, depthLimit, visited, result);
        if (!found) result.clear();
        return result;
    }

    vector<int> iterativeDeepeningDFS(int startVertex, int target, int maxDepth) {
        vector<int> result;
        for (int depth = 0; depth <= maxDepth; depth++) {
            cout << "Trying depth limit: " << depth << endl;
            result = depthLimitedSearch(startVertex, target, depth);
            if (!result.empty()) return result;
        }
        return result;
    }
};

void printTraversal(const vector<int>& traversal, const string& traversalType) {
    cout << traversalType << " traversal result: ";
    for (int vertex : traversal) {
        cout << vertex << " ";
    }
    cout << endl;
}

int main() {
    Graph* g = nullptr;
    int choice;
    int vertex1, vertex2, startVertex, targetVertex, depthLimit, maxDepth;

    do {
        cout << "\nGraph Menu:\n";
        cout << "1. Initialize Graph\n";
        cout << "2. Add Edges\n";
```

```
        cout << "3. Print Graph\n";
        cout << "4. DFS (Recursive)\n";
        cout << "5. DFS (Non-Recursive)\n";
        cout << "6. BFS\n";
        cout << "7. Depth-Limited Search (DLS)\n";
        cout << "8. Iterative Deepening DFS (IDDFS)\n";
        cout << "9. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                delete g;
                g = new Graph();
                cout << "Graph initialized." << endl;
                break;

            case 2:
                if (!g) {
                    cout << "Please initialize the graph first.\n";
                    break;
                }
                int numberOfEdges;
                cout << "How many edges do you want to add? ";
                cin >> numberOfEdges;
                cout << "Enter " << numberOfEdges << " edges (format: vertex1 vertex2):" <<
endl;
                for (int i = 0; i < numberOfEdges; ++i) {
                    cin >> vertex1 >> vertex2;
                    g->addEdge(vertex1, vertex2);
                }
                break;

            case 3:
                if (!g) { cout << "Please initialize the graph first.\n"; break; }
                g->printGraph();
                break;

            case 4:
                if (!g) { cout << "Please initialize the graph first.\n"; break; }
                cout << "Enter starting vertex for DFS (Recursive): ";
```

```
            cin >> startVertex;
            printTraversal(g->dfsRecursive(startVertex), "DFS (Recursive)");
            break;

        case 5:
            if (!g) { cout << "Please initialize the graph first.\n"; break; }
            cout << "Enter starting vertex for DFS (Non-Recursive): ";
            cin >> startVertex;
            printTraversal(g->dfsNonRecursive(startVertex), "DFS (Non-Recursive)");
            break;

        case 6:
            if (!g) { cout << "Please initialize the graph first.\n"; break; }
            cout << "Enter starting vertex for BFS: ";
            cin >> startVertex;
            printTraversal(g->bfs(startVertex), "BFS");
            break;

        case 7:
            if (!g) { cout << "Please initialize the graph first.\n"; break; }
            cout << "Enter starting vertex for DLS: ";
            cin >> startVertex;
            cout << "Enter target vertex: ";
            cin >> targetVertex;
            cout << "Enter depth limit: ";
            cin >> depthLimit;
            {
                vector<int> dlsResult = g->depthLimitedSearch(startVertex, targetVertex,
depthLimit);
                if (dlsResult.empty()) {
                    cout << "Target " << targetVertex << " not found within depth limit " <<
depthLimit << endl;
                } else {
                    printTraversal(dlsResult, "DLS Path to Target " + to_string(targetVertex));
                }
            }
            break;

        case 8:
            if (!g) { cout << "Please initialize the graph first.\n"; break; }
            cout << "Enter starting vertex for IDDFS: ";
```

```
            cin >> startVertex;
            cout << "Enter target vertex: ";
            cin >> targetVertex;
            cout << "Enter maximum depth: ";
            cin >> maxDepth;
            {
               vector<int> iddfsResult = g->iterativeDeepeningDFS(startVertex, targetVertex,
maxDepth);
               if (iddfsResult.empty()) {
                  cout << "Target " << targetVertex << " not found within max depth " <<
maxDepth << endl;
               } else {
                  printTraversal(iddfsResult, "IDDFS Path to Target " + to_string(targetVertex));
               }
            }
            break;

         case 9:
            cout << "Exiting program." << endl;
            break;

         default:
            cout << "Invalid choice. Please try again." << endl;
      }
   } while (choice != 9);

   delete g;
   return 0;
}
/*1 2
1 6
2 3
2 4
6 7
6 8
4 5
7 5
6 8*/
```