**Date: 06th January 2025**

**Experiment 1**

**AIM:** To implement DFS and BFS for traversing a graph from source node (S) to goal node (G), where source node and goal node is given by the user as an input.

**Introduction:** BFS (Breadth-First Search) and DFS (Depth-First Search) are fundamental algorithms for traversing or searching tree or graph data structures. They offer distinct approaches to exploring the nodes within these structures.

**BFS**

* **Concept:**
  + Starts at the root node and explores all neighbouring nodes at the current depth before moving to the next depth level.
  + Uses a **queue** data structure to manage the order of node visits.
* **Analogy:** Imagine exploring a maze by systematically searching each room on a floor before moving to the next floor.

**DFS**

* **Concept:**
  + Starts at the root node and explores as deeply as possible along each branch before backtracking.
  + Uses a **stack** (or recursion) to keep track of the nodes to visit.
* **Analogy:** Imagine exploring a maze by following a single path as far as possible before turning back and trying another path.

**Choosing the Right Algorithm**

The choice between BFS and DFS depends on the specific problem and the desired outcome:

* **Use BFS when:**
  + You need to find the shortest path in an unweighted graph.
  + You want to explore nodes in a breadthwise manner.
* **Use DFS when:**
  + You need to explore the entire graph or tree deeply.
  + You are dealing with problems like cycle detection or topological sorting.

|  |
| --- |
| **BFS:** |
| **Program:**  #include <iostream>  #include <vector>  #include <unordered\_map>  #include <stack>  using namespace std;  void findAllPaths(const unordered\_map<int, vector<int>> &graph, int source, int destination, vector<int> &path, vector<vector<int>> &allPaths, unordered\_map<int, bool> &visited) {  visited[source] = true;  path.push\_back(source);  if (source == destination) {  allPaths.push\_back(path);  } else {  for (int neighbor : graph.at(source)) {  if (!visited[neighbor]) {  findAllPaths(graph, neighbor, destination, path, allPaths, visited);  }  }  }  path.pop\_back();  visited[source] = false;  }  int main() {  unordered\_map<int, vector<int>> graph = {  {0, {1, 2}},  {1, {0, 2, 3}},  {2, {0, 1, 4}},  {3, {1, 4}},  {4, {2, 3}}  };  int source, destination;  cout << "Enter source node: ";  cin >> source;  cout << "Enter destination node: ";  cin >> destination;  if (graph.find(source) == graph.end() || graph.find(destination) == graph.end()) {  cout << "Source or destination node does not exist." << endl;  return 1;  }  vector<int> path;  vector<vector<int>> allPaths;  unordered\_map<int, bool> visited;  findAllPaths(graph, source, destination, path, allPaths, visited);  if (allPaths.empty()) {  cout << "No paths exist from " << source << " to " << destination << "." << endl;  } else {  cout << "All paths from " << source << " to " << destination << ":" << endl;  for (const auto &p : allPaths) {  for (size\_t i = 0; i < p.size(); ++i) {  cout << p[i];  if (i < p.size() - 1) cout << " -> ";  }  cout << endl;  }  }  return 0;  } |
| **Output:** |

|  |
| --- |
| **DFS:** |
| **Program:**  #include <iostream>  #include <vector>  #include <unordered\_map>  using namespace std;  void findAllPathsDFS(const unordered\_map<int, vector<int>> &graph, int current, int destination, vector<int> &path, vector<vector<int>> &allPaths, unordered\_map<int, bool> &visited) {  visited[current] = true;  path.push\_back(current);    if (current == destination) {  allPaths.push\_back(path);  } else {  for (int neighbor : graph.at(current)) {  if (!visited[neighbor]) {  findAllPathsDFS(graph, neighbor, destination, path, allPaths, visited);  }  }  }  path.pop\_back();  visited[current] = false;  }  void findPaths(const unordered\_map<int, vector<int>> &graph, int source, int destination) {  vector<int> path;  vector<vector<int>> allPaths;  unordered\_map<int, bool> visited;  findAllPathsDFS(graph, source, destination, path, allPaths, visited);  if (allPaths.empty()) {  cout << "No paths exist from " << source << " to " << destination << "." << endl;  } else {  cout << "All paths from " << source << " to " << destination << ":" << endl;  for (const auto &p : allPaths) {  for (size\_t i = 0; i < p.size(); ++i) {  cout << p[i];  if (i < p.size() - 1) cout << " -> ";  }  cout << endl;  }  }  }  int main() {  unordered\_map<int, vector<int>> graph = {  {0, {1, 2}},  {1, {0, 2, 3}},  {2, {0, 1, 4}},  {3, {1, 4}},  {4, {2, 3}}  };  int source, destination;  cout << "Enter source node: ";  cin >> source;  cout << "Enter destination node: ";  cin >> destination;  if (graph.find(source) == graph.end() || graph.find(destination) == graph.end()) {  cout << "Source or destination node does not exist in the graph." << endl;  return 1;  }  findPaths(graph, source, destination);  return 0;  } |
| **Output:** |

**Applications:**

**BFS (Breadth-First Search)**

* + Social Network Friend Suggestions: Finding connections between you and others.
  + GPS Navigation (walking/cycling): Finding the shortest path by exploring nearby streets first.
  + Network Broadcasting: Efficiently sending messages across a network.

**DFS (Depth-First Search)**

* + Maze Solving: Exploring one path deeply before backtracking.
  + Web Crawling: Following links on web pages to explore the internet.
  + File System Navigation: Searching for a file by exploring subdirectories deeply.

**Key Differences:**

* BFS: Explores breadthwise, finds shortest paths in unweighted graphs.
* DFS: Explores depthwise, useful for cycle detection, topological sorting, and exploring all paths.

**Questions**

**1. If a question asks for the shortest path or requires processing all nodes at a particular level / distance, you shall use \_\_\_\_\_\_\_\_\_\_\_.**

Breadth-First Search (BFS)

**2. For problems other than the category mentioned in Q1, you must use \_\_\_\_\_\_\_.**

Depth-First Search (DFS)

**3. Enhance your code to detect cycle in undirected graph:**

|  |
| --- |
| **Code:** #include <iostream>  #include <vector>  #include <unordered\_map>  #include <stack>  #include <unordered\_set>  using namespace std;  void findAllPaths(const unordered\_map<int, vector<int>> &graph, int source, int destination, vector<int> &path, vector<vector<int>> &allPaths, unordered\_map<int, bool> &visited) {      visited[source] = true;      path.push\_back(source);      if (source == destination) {          allPaths.push\_back(path);      } else {          for (int neighbor : graph.at(source)) {              if (!visited[neighbor]) {                  findAllPaths(graph, neighbor, destination, path, allPaths, visited);              }          }      }      path.pop\_back();      visited[source] = false;  }  void constructCycle(int start, int current, const unordered\_map<int, int> &parent, vector<int> &cycle) {      cycle.push\_back(start);      while (current != start) {          cycle.push\_back(current);          current = parent.at(current);      }      cycle.push\_back(start);  }  bool hasCycleUtil(const unordered\_map<int, vector<int>> &graph, int node, unordered\_map<int, bool> &visited, unordered\_map<int, int> &parent, vector<int> &cycle) {      visited[node] = true;      for (int neighbor : graph.at(node)) {          if (!visited[neighbor]) {              parent[neighbor] = node;              if (hasCycleUtil(graph, neighbor, visited, parent, cycle)) {                  return true;              }          } else if (neighbor != parent[node]) {              // Cycle detected              constructCycle(neighbor, node, parent, cycle);              return true;          }      }      return false;  }  bool hasCycle(const unordered\_map<int, vector<int>> &graph, vector<int> &cycle) {      unordered\_map<int, bool> visited;      unordered\_map<int, int> parent;      for (const auto &node : graph) {          if (!visited[node.first]) {              if (hasCycleUtil(graph, node.first, visited, parent, cycle)) {                  return true;              }          }      }      return false;  }  int main() {      unordered\_map<int, vector<int>> graph = {          {0, {1, 2}},          {1, {0, 2, 3}},          {2, {0, 1, 4}},          {3, {1, 4}},          {4, {2, 3}}      };      int source, destination;      cout << "Enter source node: ";      cin >> source;      cout << "Enter destination node: ";      cin >> destination;      // Validate source and destination      if (graph.find(source) == graph.end() || graph.find(destination) == graph.end()) {          cout << "Source or destination node does not exist in the graph." << endl;          return 1;      }      vector<int> cycle;      if (hasCycle(graph, cycle)) {          cout << "The graph contains a cycle: ";          for (int i = 0; i < cycle.size(); ++i) {              cout << cycle[i];              if (i < cycle.size() - 1) cout << " -> ";          }          cout << endl;      } else {          cout << "The graph does not contain any cycles." << endl;      }      vector<int> path;      vector<vector<int>> allPaths;      unordered\_map<int, bool> visited;      findAllPaths(graph, source, destination, path, allPaths, visited);      if (allPaths.empty()) {          cout << "No paths exist from " << source << " to " << destination << "." << endl;      } else {          cout << "All paths from " << source << " to " << destination << ":" << endl;          for (const auto &p : allPaths) {              for (size\_t i = 0; i < p.size(); ++i) {                  cout << p[i];                  if (i < p.size() - 1) cout << " -> ";              }              cout << endl;          }      }      return 0;  } |
| **Output:** |

**4. Given two integers n and m, find all the stepping numbers in range [n,m] using DFS/BFS:**

|  |
| --- |
| **Code:**  #include <iostream>  #include <queue>  #include <vector>  using namespace std;  vector<int> findSteppingNumbers(int n, int m) {      vector<int> result;      queue<int> q;      for (int i = 0; i <= 9; ++i) {          if (n <= i && i <= m) {              result.push\_back(i);          }          q.push(i);      }      while (!q.empty()) {          int curr = q.front();          q.pop();          int lastDigit = curr % 10;          int next1 = curr \* 10 + lastDigit + 1;          int next2 = curr \* 10 + lastDigit - 1;          if (0 <= lastDigit + 1 && lastDigit + 1 <= 9 && n <= next1 && next1 <= m) {              result.push\_back(next1);              q.push(next1);          }          if (0 <= lastDigit - 1 && lastDigit - 1 <= 9 && n <= next2 && next2 <= m) {              result.push\_back(next2);              q.push(next2);          }      }      return result;  }  int main() {      int n, m;      cout << "Enter n and m : \n";      cin >> n >> m;      vector<int> steppingNumbers = findSteppingNumbers(n, m);      cout << "Stepping numbers in range [" << n << ", " << m << "]: ";      for (int num : steppingNumbers) {          cout << num << " ";      }      cout << endl;      return 0;  } |
| **Output:** |