Depth First Search (DFS) algorithm

***Source:* https://github.com/rusevrosen/codehub.github.io/tree/main/Algorithms**

1. *What is DFS algorithm?*

Depth-first search (DFS) is a search algorithm that explores as far as possible along each branch before backtracking. It can be used to search a tree or graph data structure.

In DFS, the algorithm starts at the root node and explores as far as possible down each branch before backtracking. This means that it will explore the deepest possible node in the tree before exploring its siblings.

DFS has a number of applications, including finding the shortest path in a graph, solving puzzles, and testing the connectivity of a graph. It is also used in artificial intelligence and natural language processing to perform tasks such as parsing and planning.

To implement DFS in C++, you can use a stack data structure to keep track of the nodes that need to be visited. The algorithm works by pushing the root node onto the stack and then repeating the following steps:

1. Pop the top node from the stack.
2. Visit the node and mark it as visited.
3. Push all of the node's unvisited neighbors onto the stack.
4. Repeat the process until the stack is empty.

This process continues until all of the nodes in the tree have been visited.

2. *Code example.*

Depth-first search (DFS) to traverse a graph.

```cpp
#include <iostream>

#include <vector>


// Graph class with adjacency list representation

class Graph {

  int numVertices;  // Number of vertices

  std::vector<std::vector<int>> adjList;  // Adjacency list


public:

  Graph(int numVertices) {

    this->numVertices = numVertices;

    adjList.resize(numVertices);

  }


  void addEdge(int u, int v) {

    // Add an undirected edge from vertex u to vertex v

    adjList[u].push_back(v);

    adjList[v].push_back(u);

  }


  // Depth-first search (DFS) traversal of the graph

  void DFS(int start) {

    // Mark all vertices as not visited

    std::vector<bool> visited(numVertices, false);


    // Create a stack to store the vertices to be visited

    std::vector<int> stack;


    // Push the start vertex onto the stack

    stack.push_back(start);


    // Continue until the stack is empty

    while (!stack.empty()) {

      // Pop the top vertex from the stack

      int curr = stack.back();

      stack.pop_back();
// If the vertex has not been visited, visit it
      if (!visited[curr]) {
        std::cout << curr << " ";
        visited[curr] = true;
      }

      // Push all of the vertex's unvisited neighbors onto the stack
      for (int neighbor : adjList[curr]) {
        if (!visited[neighbor]) {
          stack.push_back(neighbor);
        }
      }
    }
  }
};

int main() {
  // Create a graph with 5 vertices
  Graph g(5);

  // Add edges between the vertices
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 3);
  g.addEdge(3, 4);

  // Traverse the graph using DFS, starting from vertex 0
  g.DFS(0);
  return 0;
}
```

.

3. Code explanation.

In the example, is implemented a class **`Graph`** to represent a graph with an adjacency list. The **`Graph`** class has the following methods:

- **`Graph(int numVertices)`**: This is the constructor for the **`Graph`** class, which initializes the number of vertices and creates an empty adjacency list.
- **`addEdge(int u, int v)`**: This method adds an undirected edge from vertex u to vertex v. It does this by adding v to the list of neighbors for u and u to the list of neighbors for v.
- **`DFS(int start)`**: This method performs a depth-first search (DFS) traversal of the graph, starting from the vertex start. It uses a stack data structure to keep track of the vertices that need to be visited and marks each vertex as visited once it has been visited.

In the main function is created a graph with 5 vertices and add edges between them. We then call the DFS method on the graph, starting from vertex 0. The DFS method uses a stack to store the vertices that need to be visited and repeatedly pops the top vertex from the stack, visits it, and pushes all of its unvisited neighbors onto the stack. This process continues until the stack is empty, at which point the DFS traversal is complete.

This code creates a graph with 5 vertices and adds edges between them. It then performs a depth-first search (DFS) traversal of the graph, starting from vertex 0. The output of this code will be: 0 1 2 3 4 .

This graph has 5 vertices (labeled 0 through 4) and 5 edges. It is an undirected graph, which means that the edges do not have a direction and can be traversed in either direction.

```
  0
 / \
1 - 2
    |
    3
    |
    4
```

4. Time complexity.

The time complexity of the depth-first search (DFS) algorithm implemented in the example above is O(V + E), where V is the number of vertices and E is the number of edges in the graph.

This is because the algorithm visits each vertex and each edge exactly once. Visiting a vertex takes O(1) time, and visiting an edge takes O(1) time, so the total time complexity is O(V + E).

In the worst case, the DFS algorithm will have to visit all of the vertices and edges in the graph, so the time complexity will be proportional to the size of the graph. However, in practice, the DFS algorithm may terminate early if it finds the search key before visiting all of the vertices and edges, in which case the time complexity will be lower.

I wish you happy learning,

Yours Rosen Rusev