Act 4.1 - Grafo: sus representaciones y sus recorridos.

Daniel Esparza Arizpe - A01637076

Programación de estructuras de datos y algoritmos fundamentales

Grupo 605

Tecnológico de Monterrey

Lunes 13 de noviembre 2023

Código:

```
#include <iostream>
#include <list>
#include <stack>
#include <queue>
using namespace std;

class Graph {
   private:
        int n; // Number of vertices
        list<int> * adj; // Adj list

   public:
       Graph(int n);
       void loadGraph(int V, int W);
       void DFS(int s); // Search key
       void BFS(int s);
};

Graph::Graph(int n){
   this->n = n;
   adj = new list<int>[n];
}

void Graph::loadGraph(int n, int m){
   adj[n].push_back(m);
}

void Graph::DFS(int s){
   bool *visited = new bool[n];

   for(int i = 0 ; i < n; i++){
       visited[i] = false;
   }

   stack<int> stack;
   stack.push(s);
```

```
   while (!stack.empty()){
       s = stack.top();
       stack.pop();

       if(!visited[s]){
           cout << s << " ";
           visited[s] = true;
       }

       for(list<int>::iterator i = adj[s].begin(); i != adj[s].end();
++i){
           if(!visited[*i]){
               stack.push(*i);
           }
       }

   }

}

void Graph::BFS(int s){
   bool *visited = new bool[n];

   for(int i = 0; i < n; i++){
       visited[i] = false;
   }

   queue<int> queue;
   queue.push(s);

   while (!queue.empty()){
       s = queue.front();
       queue.pop();

       if(!visited[s]){
           cout << s << " ";
           visited[s] = true;
       }
```

```
        for(list<int>::iterator i = adj[s].begin(); i != adj[s].end();
++i){
            if(!visited[*i]){
                queue.push(*i);
            }
        }
    }

}

int main() {
    //Grafo lineal:
    Graph g1(4);
    g1.loadGraph(0, 1);
    g1.loadGraph(1, 2);
    g1.loadGraph(2, 3);
    cout << "Grafo lineal: " << endl;
    g1.DFS(0);
    cout << endl;
    g1.BFS(0);
    cout << endl;

    //Grafo no conectado:
    Graph g2(6);
    g2.loadGraph(0, 1);
    g2.loadGraph(1, 2);
    g2.loadGraph(3, 4);
    g2.loadGraph(4, 5);
    cout << "Grafo no conectado: " << endl;
    g2.DFS(0);
    cout << endl;
    g2.BFS(0);
    cout << endl;

    //Grafo cíclico:
    Graph g3(4);
    g3.loadGraph(0, 1);
    g3.loadGraph(1, 2);
    g3.loadGraph(2, 3);
```

```
    g3.loadGraph(3, 0);
    cout << "Grafo ciclico: " << endl;
    g3.DFS(0);
    cout << endl;
    g3.BFS(0);
    cout << endl;

    //Grafo con múltiples aristas:
    Graph g4(4);
    g4.loadGraph(0, 1);
    g4.loadGraph(0, 2);
    g4.loadGraph(1, 2);
    g4.loadGraph(2, 0);
    g4.loadGraph(2, 3);
    g4.loadGraph(3, 3);
    cout << "Grafo con multiples aristas: " << endl;
    g4.DFS(0);
    cout << endl;
    g4.BFS(0);


}
```

Complejidad:

- La complejidad de tiempo de los métodos DFS y BFS es O(V + E), donde V es el número de vértices y E es el número de aristas en el grafo. La complejidad de espacio es O(V) para ambos métodos.

- La complejidad de tiempo del método loadGraph es O(1), al igual que la de espacio.