Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Monterrey

Act 4.1 - Grafo: sus representaciones y sus recorridos

Programación de estructuras de datos y algoritmos fundamentales (Gpo 610)

Eduardo López Benítez

Fidel Morales Briones A01198630

Monterrey, Nuevo León, a 6 de noviembre de 2023

Casos prueba:

Grafo 1:

```
cout << "Grafo 1: \n";
Graph grafo1 = loadGraph(5, 5);

cout << "BFS: \n";
grafo1.BFS(0);

cout << "\nDFS: \n";
grafo1.DFS(0);
cout << "\nMatriz de adyacencia: \n";
grafo1.toString();
```

```
Grafo 1:
Vertice incial del arco: 0
Vertice final del arco: 1
Vertice incial del arco: 1
Vertice final del arco: 2
Vertice incial del arco: 2
Vertice final del arco: 3
Vertice incial del arco: 3
Vertice final del arco: 4
Vertice incial del arco: 4
Vertice final del arco: 0
BFS:
0 1 4 2 3
DFS:
0 1 2 3 4
Matriz de adyacencia:
    0 1 2 3 4
_____________
0 | 0 1 0 0 1
1 | 1 0 1 0 0
2 | 0 1 0 1 0
3 | 0 0 1 0 1
4 | 1 0 0 1 0
```

Grafo 2:

```
cout << "\nGrafo 2: \n";
Graph grafo2 = loadGraph(5, 3);

cout << "BFS: \n";
grafo2.BFS(0);

cout << "\nDFS: \n";
grafo2.DFS(0);
cout << "\nMatriz de adyacencia: \n";
grafo2.toString();
```

```
Grafo 2:
Vertice incial del arco: 2
Vertice final del arco: 0
Vertice incial del arco: 4
Vertice final del arco: 2
Vertice incial del arco: 1
Vertice final del arco: 0
BFS:
0 1 2 4
DFS:
0 1 2 4
Matriz de adyacencia:
    0 1 2 3 4

______________
0 | 0 1 1 0 0
1 | 1 0 0 0 0
2 | 1 0 0 0 1
3 | 0 0 0 0 0
4 | 0 0 1 0 0
```

Grafo 3:

```
cout << "\nGrafo 3: \n";
Graph grafo3 = loadGraph(6, 2);

cout << "BFS: \n";
grafo3.BFS(0);

cout << "\nDFS: \n";
grafo3.DFS(0);
cout << "\nMatriz de adyacencia: \n";
grafo3.toString();
```

```
Grafo 3:
Vertice incial del arco: 0
Vertice final del arco: 5
Vertice incial del arco: 1
Vertice final del arco: 0
BFS:
0 1 5
DFS:
0 1 5
Matriz de adyacencia:
    0 1 2 3 4 5

______________
0 | 0 1 0 0 0 1
1 | 1 0 0 0 0 0
2 | 0 0 0 0 0 0
3 | 0 0 0 0 0 0
4 | 0 0 0 0 0 0
5 | 1 0 0 0 0 0
```

Grafo 4:

```
cout << "\nGrafo 4: \n";
Graph grafo4 = loadGraph(3, 3);

cout << "BFS: \n";
grafo4.BFS(0);

cout << "\nDFS: \n";
grafo4.DFS(0);
cout << "\nMatriz de adyacencia: \n";
grafo4.toString();
```

```
Grafo 4:
Vertice incial del arco: 0
Vertice final del arco: 0
Vertice incial del arco: 1
Vertice final del arco: 1
Vertice incial del arco: 2
Vertice final del arco: 2
BFS:
0
DFS:
0
Matriz de adyacencia:
    0 1 2

______________
0 | 1 0 0
1 | 0 1 0
2 | 0 0 1
```

Programa:

```
/*
Act 4.1 - Grafo: sus representaciones y sus recorridos
Fidel Morales Briones A01198630
6 de noviembre de 2023
*/

#include <iostream>
#include <vector>
#include <queue>
#include <list>
using namespace std;

class Graph {
private:
    bool** adjMatrix;
    list<int> *adjList;
    int numVertices;

public:
    /*
    * Graph, constructor de un grafo
    *
    * Se genera una lista de adyacencia y una matriz de adyacencia
    * con el número de vertices
    *
    * @param numVertices: número de vertices del grafo
    * @return: instancia un grafo
    * Complejidad tiempo: O(n^2)
    * Complejidad espacio: O(1)
    */
    Graph(int numVertices) {
    this->numVertices = numVertices;
    adjMatrix = new bool* [numVertices];
    for (int i = 0; i < numVertices; i++) {
        adjMatrix[i] = new bool[numVertices];
        for (int j = 0; j < numVertices; j++) {
            adjMatrix[i][j] = false;
        }
    }
    adjList = new list<int>[numVertices];
    }
```

```
    /*
    * addEdge, agrega un arco al grafo
    *
    * Se agrega un arco a la matriz de adyacencia y a la lista de
adyacencia
    *
    * @param i: vertice inicial del arco
    * @param j: vertice final del arco
    * @return: no tiene
    * Complejidad tiempo: O(1)
    * Complejidad espacio: O(1)
    */
    void addEdge(int i, int j) {
        adjMatrix[i][j] = true;
        adjMatrix[j][i] = true;

        adjList[i].push_back(j);
        adjList[j].push_back(i);
    }

    /*
    * BFS, búsqueda en anchura
    *
    * Se recorre el grafo en anchura utilizando una cola y se imprime
el recorrido.
    * Se utiliza un vector de los vertices visitados para no repetirlos
y una cola
    * para recorrer el grafo.
    *
    * @param start: vertice inicial del recorrido
    * @return: en la terminal, se imprimirá el recorrido
    * Complejidad tiempo: O(V+A) número de vértices más arcos
    * Complejidad espacio: O(V) número de vértices
    */
    void BFS(int start) {
        vector<bool> visited(numVertices, false);
        queue<int> q;
        q.push(start);
        visited[start] = true;
        while (!q.empty()) {
            int curr = q.front();
            q.pop();
            cout << curr << " ";
```

```
            for (int i = 0; i < numVertices; i++) {
                if (adjMatrix[curr][i] && !visited[i]) {
                    q.push(i);
                    visited[i] = true;
                }
            }
        }
    }

    /*
    * DFS, búsqueda en anchura
    *
    * Se recorre el grafo en profundidad utilizando recursión y se
imprime el recorrido.
    * Se crea un vector de los vertices visitados para no repetirlos.
    *
    * @param start: vertice inicial del recorrido
    * @return: en la terminal, se imprimirá el recorrido
    * Complejidad tiempo: O(V+A) número de vértices más arcos
    * Complejidad espacio: O(V+A) número de vértices más arcos
    */
    void DFS(int start) {
        vector<bool> visited(numVertices, false);
        dfsUtil(start, visited);
    }

    /*
    * dfsUtil, método auxiliar para el recorrido en profundidad
    *
    * Se define al vertice actual como visitado y se imprime, después
se recorre
    * el siguiente valor en la lista de adyacencia y se llama
recursivamente al método.
    *
    * @param start: vertice inicial del recorrido
    * @return: en la terminal, se imprimirá el recorrido
    * Complejidad tiempo: O(V+A) número de vértices más arcos
    * Complejidad espacio: O(n)
    */
    void dfsUtil(int curr, vector<bool>& visited) {
        visited[curr] = true;
        cout << curr << " ";
        for (int i = 0; i < numVertices; i++) {
```

```
            if (adjMatrix[curr][i] && !visited[i]) {
                dfsUtil(i, visited);
            }
        }
    }

    /*
    * toString, imprime la matriz de adyacencia
    *
    * Itera a través de los valores de la matriz de adyacencia e
imprime
    * los valores de cada fila.
    *
    * @param no tiene
    * @return: en la terminal, se imprimirá la matriz de adyacencia
    * Complejidad tiempo: O(n^2)
    * Complejidad espacio: O(1)
    */
    void toString() {
        string filaSuperior = "    ";

        for (int i = 0; i < numVertices; i++) {
            filaSuperior += to_string(i) + " ";
        }

        cout << filaSuperior << "\n_____________\n";

        for (int i = 0; i < numVertices; i++) {
            cout << i << " | ";
            for (int j = 0; j < numVertices; j++)
                cout << adjMatrix[i][j] << " ";
            cout << "\n";
        }
    }
};

/*
* loadGraph, crea un grafo dado el número de vertices y los arcos que
tendrá
*
* Se crea un grafo con el número de vértices indicado, después se
añaden los arcos
```

```
* con los vértices iniciales y finales que se indiquen con el método
addEdge.
*
* @param numVertices: número de vertices del grafo
* @param numArcos: número de arcos del grafo
* @return: un objeto grafo
* Complejidad tiempo: O(n^2)
* Complejidad espacio: O(1)
*/
Graph loadGraph(int numVertices, int numArcos) {
    Graph grafo(numVertices);
    for (int i = 0; i < numArcos; i++) {
        int inicio, fin;
        cout << "Vertice incial del arco: ";
        cin >> inicio;
        cout << "Vertice final del arco: ";
        cin >> fin;
        grafo.addEdge(inicio, fin);
    }
    return grafo;
}

int main() {

    cout << "Grafo 1: \n";
    Graph grafo1 = loadGraph(5, 5);

    cout << "BFS: \n";
    grafo1.BFS(0);

    cout << "\nDFS: \n";
    grafo1.DFS(0);
    cout << "\nMatriz de adyacencia: \n";
    grafo1.toString();

    cout << "\nGrafo 2: \n";
    Graph grafo2 = loadGraph(5, 3);

    cout << "BFS: \n";
    grafo2.BFS(0);

    cout << "\nDFS: \n";
    grafo2.DFS(0);
```

```
    cout << "\nMatriz de adyacencia: \n";
    grafo2.toString();

    cout << "\nGrafo 3: \n";
    Graph grafo3 = loadGraph(6, 2);

    cout << "BFS: \n";
    grafo3.BFS(0);

    cout << "\nDFS: \n";
    grafo3.DFS(0);
    cout << "\nMatriz de adyacencia: \n";
    grafo3.toString();

    cout << "\nGrafo 4: \n";
    Graph grafo4 = loadGraph(3, 3);

    cout << "BFS: \n";
    grafo4.BFS(0);

    cout << "\nDFS: \n";
    grafo4.DFS(0);
    cout << "\nMatriz de adyacencia: \n";
    grafo4.toString();

}
```

Bibliografía:

GeeksforGeeks. (2023a, marzo 5). *Add and remove edge in adjacency list representation of a graph*. https://www.geeksforgeeks.org/add-and-remove-edge-in-adjacency-list-representation-of-a-graph/

GeeksforGeeks. (2023b, junio 9). *Breadth first search or BFS for a graph*. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/?ref=lbp

GeeksforGeeks. (2023c, junio 9). *Depth first search or DFS for a graph*. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

Programiz. (s. f.). *Adjacency List*. https://www.programiz.com/dsa/graph-adjacency-list