Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Monterrey

Act 4.2 - Grafos: Algoritmos complementarios

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

Eduardo López Benítez

Fidel Morales Briones A01198630

Monterrey, Nuevo León, a 13 de noviembre de 2023

Casos de Prueba:

grafoApp1.txt

```
4
0 1
0 2
1 3
```

grafoApp2.txt

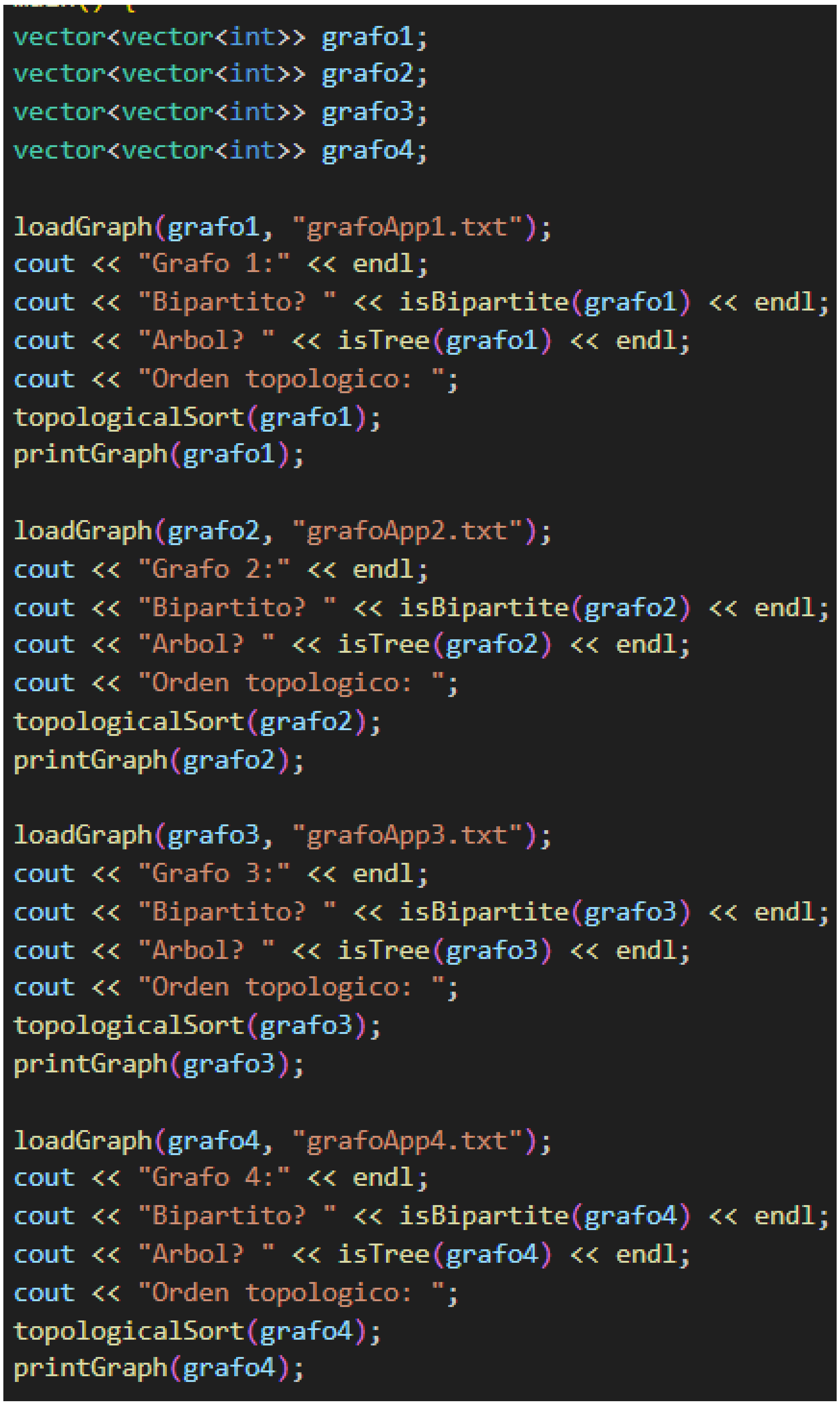
```
5
1 2
2 0
2 4
4 3
3 0
```

grafoApp3.txt

```
8
5 2
5 0
0 4
0 3
3 6
2 1
2 7
```

grafoApp4.txt

```
5
0 1
1 3
1 4
0 2
```



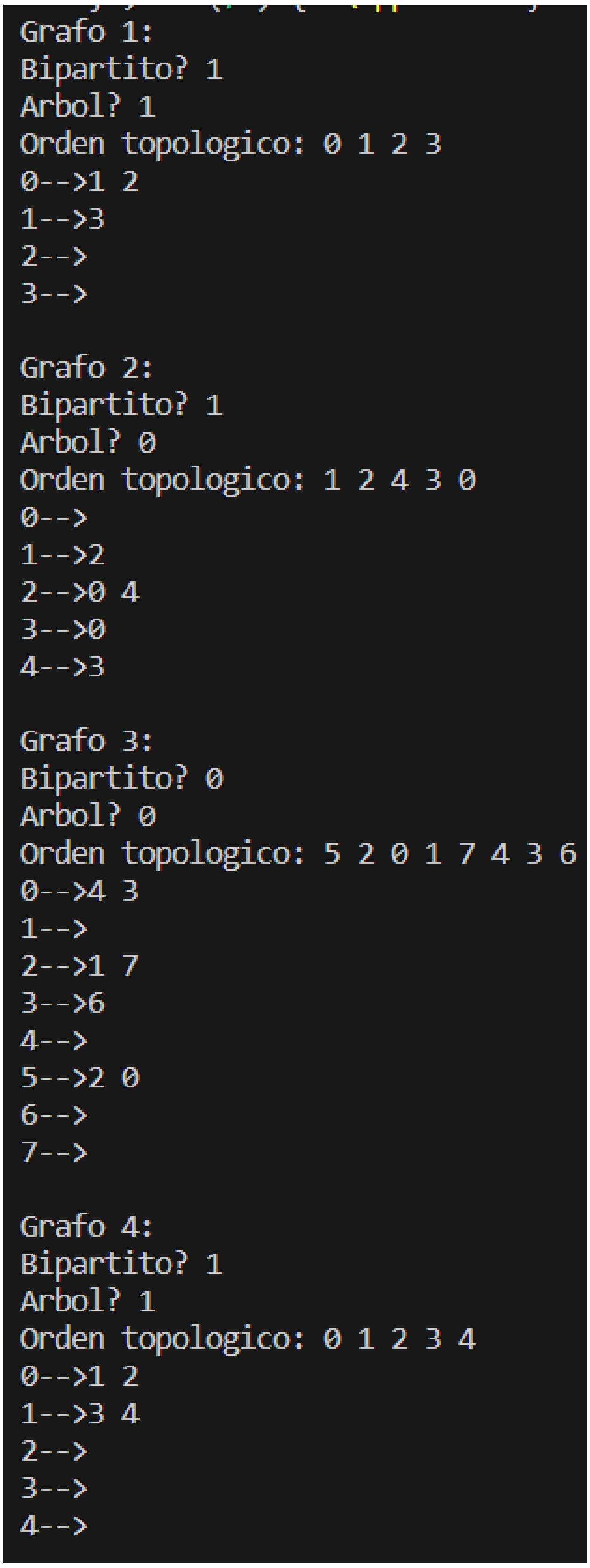
```cpp
vector<vector<int>> grafo1;
vector<vector<int>> grafo2;
vector<vector<int>> grafo3;
vector<vector<int>> grafo4;

loadGraph(grafo1, "grafoApp1.txt");
cout << "Grafo 1:" << endl;
cout << "Bipartito? " << isBipartite(grafo1) << endl;
cout << "Arbol? " << isTree(grafo1) << endl;
cout << "Orden topologico: ";
topologicalSort(grafo1);
printGraph(grafo1);

loadGraph(grafo2, "grafoApp2.txt");
cout << "Grafo 2:" << endl;
cout << "Bipartito? " << isBipartite(grafo2) << endl;
cout << "Arbol? " << isTree(grafo2) << endl;
cout << "Orden topologico: ";
topologicalSort(grafo2);
printGraph(grafo2);

loadGraph(grafo3, "grafoApp3.txt");
cout << "Grafo 3:" << endl;
cout << "Bipartito? " << isBipartite(grafo3) << endl;
cout << "Arbol? " << isTree(grafo3) << endl;
cout << "Orden topologico: ";
topologicalSort(grafo3);
printGraph(grafo3);

loadGraph(grafo4, "grafoApp4.txt");
cout << "Grafo 4:" << endl;
cout << "Bipartito? " << isBipartite(grafo4) << endl;
cout << "Arbol? " << isTree(grafo4) << endl;
cout << "Orden topologico: ";
topologicalSort(grafo4);
printGraph(grafo4);
```



```
Grafo 1:
Bipartito? 1
Arbol? 1
Orden topologico: 0 1 2 3
0-->1 2
1-->3
2-->
3-->

Grafo 2:
Bipartito? 1
Arbol? 0
Orden topologico: 1 2 4 3 0
0-->
1-->2
2-->0 4
3-->0
4-->3

Grafo 3:
Bipartito? 0
Arbol? 0
Orden topologico: 5 2 0 1 7 4 3 6
0-->4 3
1-->
2-->1 7
3-->6
4-->
5-->2 0
6-->
7-->

Grafo 4:
Bipartito? 1
Arbol? 1
Orden topologico: 0 1 2 3 4
0-->1 2
1-->3 4
2-->
3-->
4-->
```

Código:

```
/*
Act 4.2 - Grafos: Algoritmos complementarios
Fidel Morales Briones A01198630
13 de noviembre de 2023
*/

#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <queue>
using namespace std;

/*
* addEdge, agrega una arista al grafo dirigido
*
* Se agrega una arista a la lista de adyacencia
*
* @param i: vertice inicial de la arista
* @param j: vertice final de la arista
* @param adjList: lista de adyacencia
* @return: no tiene
* Complejidad tiempo: O(1)
* Complejidad espacio: O(1)
*/
void addEdge(int u, int v, vector<vector<int>> &adjList) {
    adjList[u].push_back(v);
}

/*
* printGraph, imprime la lista de adyacencia
*
* Itera a través de los valores de la lista de ayacencia e imprime
* cada vertice y los vertices a los que está conectado
*
* @param adjList: lista de adyacencia
* @return: en la terminal, se imprimirá la lista de adyacencia
* Complejidad tiempo: O(v + e)
* Complejidad espacio: O(1)
*/
void printGraph(vector<vector<int>> adjList) {
```

```cpp
    for (int i = 0; i < adjList.size(); i++) {
        cout << i << "-->";
        for (int j = 0; j < adjList[i].size(); j++) {
            cout << adjList[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

/*
* loadGraph, se carga el grafo de un archivo de texto
*
* Se carga un archivo de texto  y se lee línea por línea. Se obtiene el
número de vertices y se cambia el tamaño
* de la lista de adyacencia. Se obtienen las aristas y se agregan a la
lista de adyacencia
*
* @param adjList: lista de adyacencia a la que se agregan las aristas
* @param filename: nombre del archivo de texto
* @return: no tiene
* Complejidad tiempo: O(v + e)
* Complejidad espacio: O(1)
*/
void loadGraph(vector<vector<int>> &adjList, string filename) {
    ifstream file(filename);
    string line;
    string u, v;

    getline(file, line, '\n');
    adjList.resize(stoi(line));

    while (getline(file, u, ' ')) {
        getline(file, v, '\n');
        addEdge(stoi(u), stoi(v), adjList);
    }

}

/*
* isBipartiteUtil, método auxiliar para determinar si el grafo es
bipartito
*
```

```
* Se realiza una BFS para recorrer el grafo y se asigna un color a cada
vertice. Si un vertice adyacente
* tiene el mismo color que el vertice actual, entonces el grafo no es
bipartito. Y si no tiene color, se le asigna
* el color opuesto al vertice actual.
*
* @param ajdList: lista de adyacencia del grafo
* @param src: vertice inicial para realizar la BFS
* @param color: vector de colores para cada vertice
* @return: bool que indica si el grafo es bipartito o no
* Complejidad tiempo: O(v + e)
* Complejidad espacio: O(v)
*/
bool isBipartiteUtil(vector<vector<int>> &adjList, int src, vector<int>
&color) {
    color[src] = 1;
    queue<int> q;
    q.push(src);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v : adjList[u]) {
            if (color[v] == -1) {
                color[v] = 1 - color[u];
                q.push(v);
            }
            else if (color[v] == color[u]) {
                return false;
            }
        }
    }
    return true;
}

/*
* isBipartite, determina si el grafo es bipartito
*
* Por cada vertice del grafo, se llama al método auxiliar para
determinar si el grafo es bipartito.
*
* @param adjList: lista de adyacencia del grafo
* @return: bool que indica si el grafo es bipartito o no
* Complejidad tiempo: O(v + e)
* Complejidad espacio: O(v)
*/
bool isBipartite(vector<vector<int>> &adjList) {
    vector<int> color(adjList.size(), -1);

    for (int i = 0; i < adjList.size(); i++) {
        if (color[i] == -1) {
            if (isBipartiteUtil(adjList, i, color) == false) {
                return false;
            }
        }
    }
    return true;
}

/*
* isTree, determina si el grafo es un árbol
*
* Se realiza una BFS para recorrer el grafo. Se utiliza un vector de
visitados para almacenar
* los vertices visitados. Se revisa si el vertice actual ya fue
visitado, si no, se agrega a la cola
* y se marca como visitado. Si el vertice adyacente ya fue visitado y
no es el mismo vertice actual,
* entonces el grafo no es un árbol. También se revisa que todos los
vertices hayan sido visitados.
*
* @param adjList: lista de adyacencia del grafo
* @return: bool que indica si el grafo es un árbol o no
* Complejidad tiempo: O(v + e)
* Complejidad espacio: O(v)
*/
bool isTree(vector<vector<int>> &adjList) {
    vector<bool> visited(adjList.size(), false);
    queue<int> q;
    q.push(0);
    visited[0] = true;

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : adjList[u]) {
            if (!visited[v] && v != u) {
                visited[v] = true;
                q.push(v);
            }
            else {
                return false;
            }
        }
    }

    for (int i = 0; i < visited.size(); i++) {
        if (!visited[i]) {
            return false;
        }
    }
    return true;
}

/*
* topologicalSort, ordenamiento topológico del grafo
*
* Se crea un vector de grados de entrada para cada vértice, se recorre la lista de adyacencia para
* encontrar el grado de entrada de cada vértice. En el ciclo while se revisa cuáles
* vértices tienen grado 0 y se agregan a la cola para después eliminarlos del grafo, se reduce el grado de entrada
* de los vértices adyacentes y se repite el proceso hasta que la cola esté vacía. Finalmente se verifica si existe un
* ciclo en el grafo.
*
* @param adjList: lista de adyacencia del grafo
* @return: se imprime el orden topológico del grafo en la terminal
* Complejidad tiempo: O(v + e)
* Complejidad espacio: O(v)
*/
void topologicalSort(vector<vector<int>> &adjList) {
    vector<int> in_degree(adjList.size(), 0);

    for (int u = 0; u < adjList.size(); u++) {
        for (int v : adjList[u]) {
```

```cpp
            in_degree[v]++;
        }
    }

    queue<int> q;
    for (int i = 0; i < adjList.size(); i++) {
        if (in_degree[i] == 0) {
            q.push(i);
        }
    }

    int count = 0;
    vector<int> top_order;

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        top_order.push_back(u);

        for (int v : adjList[u]) {
            if (--in_degree[v] == 0) {
                q.push(v);
            }
        }
        count++;
    }

    if (count != adjList.size()) {
        cout << "Hay un ciclo en el grafo" << endl;
    }
    else {
        for (int i = 0; i < top_order.size(); i++) {
            cout << top_order[i] << " ";
        }
        cout << endl;
    }
}

int main() {
    vector<vector<int>> grafo1;
    vector<vector<int>> grafo2;
    vector<vector<int>> grafo3;
    loadGraph(grafo1, "grafoApp1.txt");
    cout << "Grafo 1:" << endl;
    cout << "Bipartito? " << isBipartite(grafo1) << endl;
    cout << "Arbol? " << isTree(grafo1) << endl;
    cout << "Orden topologico: ";
    topologicalSort(grafo1);
    printGraph(grafo1);

    loadGraph(grafo2, "grafoApp2.txt");
    cout << "Grafo 2:" << endl;
    cout << "Bipartito? " << isBipartite(grafo2) << endl;
    cout << "Arbol? " << isTree(grafo2) << endl;
    cout << "Orden topologico: ";
    topologicalSort(grafo2);
    printGraph(grafo2);

    loadGraph(grafo3, "grafoApp3.txt");
    cout << "Grafo 3:" << endl;
    cout << "Bipartito? " << isBipartite(grafo3) << endl;
    cout << "Arbol? " << isTree(grafo3) << endl;
    cout << "Orden topologico: ";
    topologicalSort(grafo3);
    printGraph(grafo3);

    return 0;
}
```

Bibliografía:

GeeksforGeeks. (2023, 6 junio). *KahN s algorithm for Topological sorting*. https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/

Sryheni, S. (2022, 3 septiembre). *Determining whether a directed or undirected graph is a tree*. Baeldung. https://www.baeldung.com/cs/determine-graph-is-tree

takeUforward. (2022, 6 septiembre). Bipartite Graph | BFS Implementation. *takeUforward*. https://takeuforward.org/graph/bipartite-graph-bfs-implementation/

WilliamFiset. (2020, 29 agosto). *Topological sort | Kahn's Algorithm | Graph Theory* [Vídeo]. YouTube. https://www.youtube.com/watch?v=cIBFEhD77b4