César Silva - A01252916

Act 4.1 - Grafo: sus representaciones y sus recorridos

Repositorio en github

Casos Prueba
*** Los casos prueba están hechos con grafos no dirigidos.**
*** Para los casos prueba se utilizará el mismo input con nodos iniciales de recorrido variantes entre casos.**
Input

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

Caso 1
Nodo Inicial = 0

```
Matriz de adyacencia:
0 1 1 0 0
1 0 1 1 0
1 1 0 1 0
0 1 1 0 1
0 0 0 1 0

Lista de adyacencia:
Vertice 0: 1 2
Vertice 1: 0 3 2
Vertice 2: 0 3 1
Vertice 3: 1 4 2
Vertice 4: 3

DFS :
0 1 3 4 2

BFS:
0 1 2 3 4
```

Caso 2

Nodo inicial = 2

```
Matriz de adyacencia:
0 1 1 0 0
1 0 1 1 0
1 1 0 1 0
0 1 1 0 1
0 0 0 1 0

Lista de adyacencia:
Vertice 0: 1 2
Vertice 1: 0 3 2
Vertice 2: 0 3 1
Vertice 3: 1 4 2
Vertice 4: 3

DFS :
2 0 1 3 4

BFS:
2 0 3 1 4
```

Caso 3

Nodo inicial = 3

```
Matriz de adyacencia:
0 1 1 0 0
1 0 1 1 0
1 1 0 1 0
0 1 1 0 1
0 0 0 1 0

Lista de adyacencia:
Vertice 0: 1 2
Vertice 1: 0 3 2
Vertice 2: 0 3 1
Vertice 3: 1 4 2
Vertice 4: 3

DFS :
3 1 0 2 4

BFS:
3 1 4 2 0
```

Caso 4

Nodo inicial = 4

```
Matriz de adyacencia:
0 1 1 0 0
1 0 1 1 0
1 1 0 1 0
0 1 1 0 1
0 0 0 1 0

Lista de adyacencia:
Vertice 0: 1 2
Vertice 1: 0 3 2
Vertice 2: 0 3 1
Vertice 3: 1 4 2
Vertice 4: 3

DFS :
4 3 1 0 2

BFS:
4 3 1 2 0
```

Complejidades Temporales

- loadGraph

```
void loadGraph(int n, vector<vector<int>>& adjacencyMatrix, vector<list<int>>&
adjacencyList, ifstream& inputFile) {
   adjacencyMatrix.assign(n, vector<int>(n, 0));

   int i, j;
   while (inputFile >> i >> j) {
       adjacencyMatrix[i][j] = 1;
       adjacencyMatrix[j][i] = 1;

       adjacencyList[i].push_back(j);
       adjacencyList[j].push_back(i);
   }
}
```

En cuanto a la matriz de adyacencia, la complejidad temporal de la asignación de valores por sí sola es constante, es decir, O(1). Sin embargo, al realizar esta asignación para cada arco de adyacencia entre vértices dentro del bucle while, obtenemos una complejidad temporal de O(n),

donde n representa la cantidad total de arcos. De manera similar, al examinar la lista de adyacencia, la operación de inserción de elementos es constante O(1). No obstante, dado que realizamos esta operación para cada arco, obtenemos una complejidad temporal de O(n). En conjunto, la complejidad temporal total se resume como O(n) + O(n) = O(n), donde n representa la cantidad de arcos que indican la adyacencia entre vértices.

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	0	0
v_2	1	0	1	1	0
v_3	1	1	0	0	1
v_4	0	1	0	0	1
v_5	0	0	1	1	0

v_1	v_2, v_4
v_2	v_3, v_5
v_3	v_2, v_4
v_4	v_1, v_3, v_5
v_5	v_2, v_4

- DFS

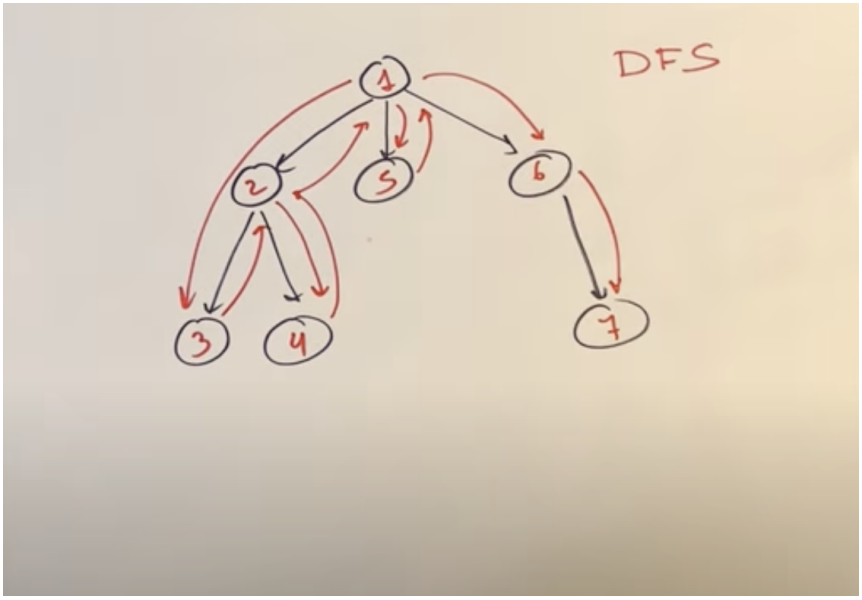
```
void DFSUtil(int vertex, vector<bool>& visited, const vector<list<int>>&
adjacencyList) {
   visited[vertex] = true;
   cout << vertex << " ";

   for (const auto& neighbor : adjacencyList[vertex]) {
       if (!visited[neighbor]) {
           DFSUtil(neighbor, visited, adjacencyList);
       }
   }
}
```

En este algoritmo, cada vértice es visitado una sola vez mediante el uso de una bandera booleana que indica si un vértice ha sido visitado, permitiendo así determinar cuándo no es necesario visitarlo nuevamente. La contribución de esta operación al tiempo de ejecución es por lo tanto, O(V), donde V representa el número de vértices en el grafo.

En el caso de un grafo no dirigido, cada arco conecta dos vértices. En consecuencia, los arcos se examinan para verificar si los vértices vecinos ya han sido visitados. En el peor escenario, sería necesario examinar todos los arcos, resultando en una complejidad de O(E), donde E es la cantidad de arcos en el grafo.

La complejidad temporal total de la función se obtiene sumando las contribuciones de los vértices y los arcos: O(V) + O(E) = O(V + E). Esto refleja que el tiempo de ejecución del algoritmo está proporcionalmente relacionado con el número de vértices y arcos en el grafo.



- BFS

```
void BFS(const vector<list<int>>& adjacencyList, int initialNode) {
    int n = adjacencyList.size();
    vector<bool> visited(n, false);

    // creación de queue para BFS
    queue<int> bfsQueue;

    // marcamos el nodo visitado
    visited[initialNode] = true;
    bfsQueue.push(initialNode);

    cout << "\nBFS: " << endl;

    while (!bfsQueue.empty()) {
        // se saca el vertice y es impreso
        int currentVertex = bfsQueue.front();
        bfsQueue.pop();
        cout << currentVertex << " ";

        for (const auto& neighbor : adjacencyList[currentVertex]) {
            if (!visited[neighbor]) {
```

```
                visited[neighbor] = true;
                bfsQueue.push(neighbor);
            }
        }
    }

    cout << endl;
}
```

Al igual que en el caso del Depth-First Search (DFS), en el peor escenario del Breadth-First Search (BFS) debemos visitar cada vértice y explorar cada arco, lo cual resulta en una complejidad temporal de O(V + E), donde V representa el número de vértices y E el número de arcos en el grafo.