Reflexión Actividad 4.1 Grafo: sus representaciones y sus recorridos

Héctor Robles Villarreal A01634105

Programación de estructuras de datos y algoritmos fundamentales

Carlos Augusto Ventura Molina

Grupo 11

Tecnológico de Monterrey Campus Guadalajara

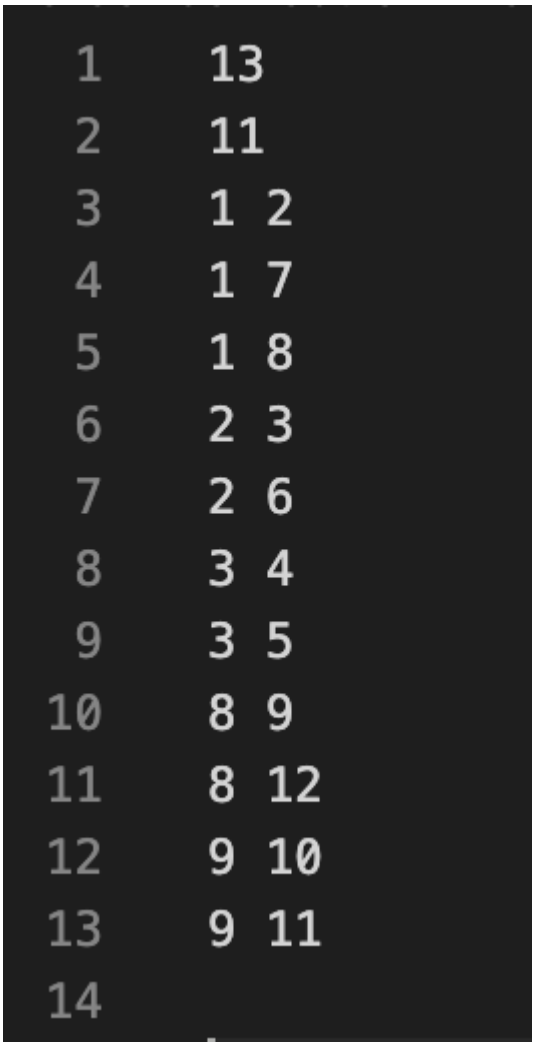
Sábado 13 de Noviembre de 2021

Reflexión Actividad 4.1 Grafo: sus representaciones y sus recorridos

Para esta actividad se implementó en el código del recorrido de los grafos, una función llamada loadGraph que cargara un grafo a partir de un archivo txt y que posteriormente llamara a las funciones de los recorridos tanto de anchura como de profundidad para desplegar los datos del grafo.

```
void Graph::loadGraph(string archivo){
    ifstream ifs;
    vector<Edge> aristas;
    ifs.open("grafo.txt");
    string line;
    int nodos;
    int arcos;
    ifs >> nodos;
    ifs >> arcos;
    Edge arista;
    int source;
    int dest;
    int cont = 0;
    aristas.resize(nodos);
    while(ifs >> source >> dest){
        arista.src = source;
        arista.dest = dest;
        aristas[cont] = arista;
        cont++;
    }
    ifs.close();
    Graph graph(aristas,nodos);
    vector<bool> discovered(nodos);
    cout << "\nRecorrido del grafo por profundidad (DFS): " << endl;
    for (int i = 0; i < nodos; i++)
        if (discovered[i] == false)
            Graph::DFS(graph, i, discovered);
    cout << std::endl;
    vector<bool> discoveredB(nodos, false);
    cout << "\nRecorrido del grafo por anchura (BFS):" << endl;
    for (int i = 0; i < nodos; i++) {
        if (discoveredB[i] == false) {
            Graph::BFS(graph, i, discoveredB);
        }
    }
    cout << endl << endl;
```

Como se puede ver, primero se abre el archivo de texto y se guarda el número de nodos y el número de aristas, posteriormente, guarda las conexiones entre nodos. Este es el archivo de texto:



El primer dato representa el número de nodos, el segundo el número de aristas o arcos y los demás datos son las conexiones entre los vertices.

Después de que se cargan los datos del grafo, se manda a llamar a la función de recorrido por profundidad:

```cpp
void Graph::DFS(Graph const& graph, int v, vector<bool>& discovered){//Time complexity O(n) Space complexity O(h) h = altura
    discovered[v] = true;
    cout << v << " ";
    // do for every edge (v -> u)
    for (int u : graph.adjList[v])
    {
        if (!discovered[u])
            Graph::DFS(graph, u, discovered);
    }
}
```

que básicamente recibe una lista de adyacencia de booleanos que representan los nodos que ya se han descubierto, se va cambiando esta lista conforme se visitan los nodos y se van imprimiendo los datos del grafo. Su complejidad en tiempo de ejecución es de O(n) ya que cada nodo se visita solo una vez, su complejidad espacial puede variar de grafo a grafo por lo que es de O(h) donde h representa la altura del grafo.

Después de llamar a la función de recorrido por profundidad se llama a la de recorrido por anchura:

```cpp
void Graph::BFS(Graph const& graph, int v, vector<bool>& discovered){ //Time complexity O(n) Space complexity O(w) w = anchura
    queue<int> q;
    discovered[v] = true;
    q.push(v);
    while (!q.empty()){
        v = q.front();
        q.pop();
        cout << v << " ";
        // do for every edge (v -> u)
        for (int u : graph.adjList[v])
            if (!discovered[u]){
                discovered[u] = true;
                q.push(u);
            }
    }
}
```

que usa una cola para ir guardando los nodos que ya se descubrieron, usando también una lista de adyacencia. Su complejidad en tiempo de ejecución también es de O(n) porque cada nodo se visita también solo una vez y su complejidad de espacio depende de la anchura del grafo por lo que es de O(w) donde w es la anchura.

Esta es la salida en el main:

```
Recorrido del grafo por profundidad (DFS):
0 1 2 3 4 5 6 7 8 9 10 11 12

Recorrido del grafo por anchura (BFS):
0 1 2 7 8 3 6 9 12 4 5 10 11
```