INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Estructuras de Datos y Algoritmos

Act 4.2 - Grafos: Algoritmos complementarios

lsCésar Alán Silva Ramos

25/11/23

Repositorio en Github

Complejidades computacionales

- **loadGraph**

```
void Grafo::loadGraph(int n, int m, std::vector<std::pair<int, int>>& edges) {
   this->V = n; // O(1)
   matrizAdj.resize(V, std::vector<int>(V, 0)); // O(V^2)
   for (auto edge : edges) { // O(E)
       int v = edge.first;
       int w = edge.second;
       matrizAdj[v][w] = 1; // O(1)
       matrizAdj[w][v] = 1; // O(1)
       listaAdj[v].push_back(w); // O(1)
       listaAdj[w].push_back(v); // O(1)
   }
}
```

En cuanto a la complejidad **espacial** de la función loadGraph, esta se caracteriza por **O(V + E)**. Esto se debe a que se necesitan V elementos para representar los vértices y E elementos para representar las aristas dentro de las listas de adyacencia.

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, 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.

- **isTree**

```
bool Grafo::isTree() {
   std::vector<bool> visited(V, false); // O(V)
   if (isCyclicUtil(0, visited, -1)) // O(V + E)
       return false;

   for (int u = 0; u < V; u++) // O(V)
       if (!visited[u]) // O(1)
           return false;

   return true;
}
```

```
bool Grafo::isCyclicUtil(int v, std::vector<bool>& visited, int parent) {
   visited[v] = true; // O(1)

   for (auto i : listaAdj[v]) { // O(E/V)
       if (!visited[i]) { // O(1)
           if (isCyclicUtil(i, visited, v)) // O(V + E)
               return true;
       }
       else if (i != parent) // O(1)
           return true;
   }
   return false;
}
```

La complejidad temporal de esta función se ve mayormente influenciada por la función, isCyclicUtil, en la cual cada llamada recursiva explora vértices y aristas adyacentes. La complejidad temporal de esta exploración depende directamente de la cantidad de vértices y

aristas en el grafo. En el escenario menos favorable, donde cada arista y vértice se examina una vez, la complejidad **temporal** que obtenemos es **O(V + E)**.

En cuanto a la complejidad **espacial**, tenemos **O(V)**. Cada vértice visitado es rastreado para determinar si el grafo constituye un árbol representado, y esta información es crucial para seguir examinando los vértices y así saber si estamos tratando con un árbol.

- **topologicalSort**

```
void Grafo::topologicalSort() {
   std::vector<bool> visited(V, false); // O(V)
   std::stack<int> Stack; // O(1)

   for (int i = 0; i < V; i++) // O(V)
       if (visited[i] == false) // O(1)
           topologicalSortUtil(i, visited, Stack); // O(V + E)

   while (Stack.empty() == false) { // O(V)
       std::cout << Stack.top() << " "; // O(1)
       Stack.pop(); // O(1)
   }
}

void Grafo::topologicalSortUtil(int v, std::vector<bool>& visited, std::stack<int>&
Stack) {
   visited[v] = true; // O(1)

   for (auto i : listaAdj[v]) // O(E/V)
       if (!visited[i]) // O(1)
           topologicalSortUtil(i, visited, Stack); // O(V + E)

   Stack.push(v); // O(1)
}
```

La complejidad **temporal** de esta función es **O(V + E)** en el peor de los casos, ya que se explorarán todos los vértices y aristas durante el recorrido ordenado del grafo. Respecto a la complejidad **espacial**, es **O(V)**. Debido al uso de un vector para rastrear los vértices visitados durante el recorrido topológico, y a una pila que colabora almacenando los vértices en este orden.

- **bipartiteGraph**

```
bool Grafo::bipartiteGraph() {
    std::vector<int> colorArr(V, -1); // O(V)

    for (int i = 0; i < V; i++) // O(V)
        if (colorArr[i] == -1) // O(1)
            if (isBipartiteUtil(i, colorArr) == false) // O(V + E)
                return false;

    return true;
}

bool Grafo::isBipartiteUtil(int src, std::vector<int>& colorArr) {
    colorArr[src] = 1; // O(1)

    std::queue<int> q; // O(1)
    q.push(src); // O(1)

    while (!q.empty()) { // O(V)
        int u = q.front(); // O(1)
        q.pop(); // O(1)

        for (auto v : listaAdj[u]) { // O(E/V)
            if (colorArr[v] == -1) { // O(1)
                colorArr[v] = 1 - colorArr[u]; // O(1)
```

```
                q.push(v); // O(1)
            }
            else if (colorArr[v] == colorArr[u]) // O(1)
                return false;
        }
    }
    return true;
}
```

La complejidad **temporal** de la función para verificar si un grafo es bipartito es **O (V+E)**, ya que se realiza un recorrido que analiza cada vértice y arista. Esto se traduce en O(V+E), donde V representa la cantidad de vértices y E la cantidad de aristas en el grafo. Esta complejidad está directamente relacionada con el tamaño del grafo. La siguiente fase para determinar la bipartición se basa en la técnica de colores, donde no pueden existir dos vértices adyacentes del mismo color en el grafo. Por ende, se requiere un rastreo de los colores asignados a cada vértice durante el proceso de bipartición, lo que resulta en una complejidad **espacial** de **O(V)** por el almacenamiento del rastreo de estos.

Casos prueba

Caso de prueba 4

```
// cuarto grafo: no árbol y bipartito
//     4
//     |
// 0---1---5
// |   |
// 3---2
//
//
```

Output

```
Matriz de adyacencia:
0: 0 1 0 1 0 0
1: 1 0 1 0 1 1
2: 0 1 0 1 0 0
3: 1 0 1 0 0 0
4: 0 1 0 0 0 0
5: 0 1 0 0 0 0

Lista de adyacencia:
0: 3 1
1: 4 2 0 5
2: 1 3
3: 2 0
4: 1
5: 1
El grafo 4 no es un árbol
El grafo 4 sí es bipartito
Orden topológico del grafo 4: 0 3 2 1 5 4
```

Caso de prueba 3

```
// tercer grafo: árbol y bipartito
// 0---1---2---3 (árbol lineal) :)
```

Output

```
Matriz de adyacencia:
0: 0 1 0 0
1: 1 0 1 0
2: 0 1 0 1
3: 0 0 1 0

Lista de adyacencia:
0: 1
1: 0 2
2: 1 3
3: 2
El grafo 3 sí es un árbol
El grafo 3 sí es bipartito
Orden topológico del grafo 3: 0 1 2 3
```

Caso de prueba 2

```
// segundo grafo: no árbol y no bipartito
// 0---1---2---3
// |           |
// 4------6-----5
```

Output

```
Matriz de adyacencia:
0: 0 1 0 0 1 0 0
1: 1 0 1 0 0 0 0
2: 0 1 0 1 0 0 0
3: 0 0 1 0 0 1 0
4: 1 0 0 0 0 0 1
5: 0 0 0 1 0 0 1
6: 0 0 0 0 1 1 0

Lista de adyacencia:
0: 1 4
1: 0 2
2: 1 3
3: 2 5
4: 6 0
5: 3 6
6: 5 4
El grafo 2 no es un árbol
El grafo 2 no es bipartito
Orden topológico del grafo 2: 0 1 2 3 5 6 4
```

Caso de prueba 1

```
// primer grafo: árbol y bipartito
// 4---3---2---1---0
```

Output

```
Matriz de adyacencia:
0: 0 1 0 0 0
1: 1 0 1 0 0
2: 0 1 0 1 0
3: 0 0 1 0 1
4: 0 0 0 1 0

Lista de adyacencia:
0: 1
1: 2 0
2: 3 1
3: 4 2
4: 3
El grafo 1 sí es un árbol
El grafo 1 sí es bipartito
Orden topológico del grafo 1: 0 1 2 3 4
```