GRAFOS en C++:

Índice

Introducción a los grafos en C++

Un grafo **es una estructura de datos** que consiste en un conjunto de **nodos (también llamados vértices)** y un conjunto de **aristas** que conectan pares de nodos. Los grafos pueden ser:

Dirigidos: Las aristas tienen una dirección, es decir, van de un nodo a otro específico.
No dirigidos: Las aristas no tienen dirección, es decir, conectan dos nodos en ambos sentidos.

Terminología básica:

Vértices (nodos): Los puntos o entidades en el grafo.
Aristas (edges): Las conexiones entre los vértices.
Grado: El número de aristas que conectan un vértice (en grafos dirigidos, se distingue entre grado de entrada y grado de salida).

Representación de un grafo

Los grafos pueden representarse de varias maneras, pero las dos más comunes en C++ son:

1.-Matriz de adyacencia

Una matriz de adyacencia es una matriz 2D donde cada entrada (i, j) indica si hay una arista entre los vértices i y j.

Ventajas: Fácil de implementar, rápida para comprobar la existencia de aristas.
Desventajas: Ocupa mucho espacio para grafos dispersos (es decir, grafos con pocas aristas en comparación con el número de nodos).

```
#include <iostream>
#include <vector>

int main() {
    int numeroDeNodos = 5;
      std::vector<std::vector<int>>  matrizAdyacencia(numeroDeNodos,  std::vector<int>(numeroDeNodos,
0));

        // Insertar una arista entre el nodo 0 y el nodo 1
        matrizAdyacencia[0][1] = 1;
        matrizAdyacencia[1][0] = 1; // Si es no dirigido

        return 0;
}
```

2.-Lista de adyacencia

Una lista de adyacencia utiliza un array (o vector) de listas, donde cada lista almacena los nodos adyacentes a un nodo dado.

Ventajas: Más eficiente en términos de espacio para grafos dispersos.
Desventajas: Puede ser más lenta para comprobar la existencia de aristas.

```cpp
#include <iostream>
#include <vector>
#include <list>

int main() {
	int numeroDeNodos = 5;
	std::vector<std::list<int>> listaAdyacencia(numeroDeNodos);

	// Insertar una arista entre el nodo 0 y el nodo 1
	listaAdyacencia[0].push_back(1);
	listaAdyacencia[1].push_back(0); // Si es no dirigido

	return 0;
}
```

Creación de un grafo

Para crear un grafo, primero debes decidir qué representación utilizarás. A continuación se muestra cómo crear un grafo usando tanto una matriz de adyacencia como una lista de adyacencia.

Usando matriz de adyacencia

```cpp
std::vector<std::vector<int>> crearGrafoConMatriz(int numeroDeNodos) {
	return std::vector<std::vector<int>>(numeroDeNodos, std::vector<int>(numeroDeNodos, 0));
}
```

Usando lista de adyacencia

cpp

```cpp
std::vector<std::list<int>> crearGrafoConLista(int numeroDeNodos) {
	return std::vector<std::list<int>>(numeroDeNodos);
}
```

Insertar aristas en un grafo

Usando matriz de adyacencia

```cpp
void insertarAristaMatriz(std::vector<std::vector<int>>& matriz, int u, int v) {
	matriz[u][v] = 1;
	matriz[v][u] = 1; // Si es no dirigido
}
```

Usando lista de adyacencia

```cpp
void insertarAristaLista(std::vector<std::list<int>>& lista, int u, int v) {
	lista[u].push_back(v);
	lista[v].push_back(u); // Si es no dirigido
}
```

Recorrer un grafo

Búsqueda en profundidad (DFS)

La búsqueda en profundidad (DFS) explora tanto como sea posible a lo largo de cada rama antes de retroceder.

```cpp
void DFSUtil(int v, std::vector<bool>& visitado, const std::vector<std::list<int>>& listaAdyacencia)
{
	visitado[v] = true;
	std::cout << v << " ";

	for (auto i : listaAdyacencia[v]) {
		if (!visitado[i]) {
			DFSUtil(i, visitado, listaAdyacencia);
		}
	}
}

void DFS(int v, const std::vector<std::list<int>>& listaAdyacencia) {
	std::vector<bool> visitado(listaAdyacencia.size(), false);
	DFSUtil(v, visitado, listaAdyacencia);
}
```

Búsqueda en anchura (BFS)

La búsqueda en anchura (BFS) explora todos los nodos a una distancia d antes de pasar a la distancia d+1.

```
#include <queue>

void BFS(int v, const std::vector<std::list<int>>& listaAdyacencia) {
        std::vector<bool> visitado(listaAdyacencia.size(), false);
        std::queue<int> cola;
        visitado[v] = true;
        cola.push(v);

        while (!cola.empty()) {
                v = cola.front();
                std::cout << v << " ";
                cola.pop();

                for (auto i : listaAdyacencia[v]) {
                        if (!visitado[i]) {
                                visitado[i] = true;
                                cola.push(i);
                        }
                }
        }
}
```

Eliminar aristas en un grafo

Usando matriz de adyacencia

```
void eliminarAristaMatriz(std::vector<std::vector<int>>& matriz, int u, int v) {
        matriz[u][v] = 0;
        matriz[v][u] = 0; // Si es no dirigido
}
```

Usando lista de adyacencia

```
void eliminarAristaLista(std::vector<std::list<int>>& lista, int u, int v) {
        lista[u].remove(v);
        lista[v].remove(u); // Si es no dirigido
}
```

Ejemplo completo de grafo

De un grafo utilizando una lista de adyacencia. Este ejemplo cubre la creación del grafo, la inserción y eliminación de aristas, así como las búsquedas en profundidad (DFS) y en anchura (BFS).

```
#include <iostream>
#include <vector>
#include <list>
#include <queue>

// Función para insertar una arista en un grafo representado por lista de adyacencia

void insertarAristaLista(std::vector<std::list<int>>& lista, int u, int v) {
        lista[u].push_back(v);
        lista[v].push_back(u); // Si es un grafo no dirigido
}

// Función para eliminar una arista en un grafo representado por lista de adyacencia

void eliminarAristaLista(std::vector<std::list<int>>& lista, int u, int v) {
        lista[u].remove(v);
        lista[v].remove(u); // Si es un grafo no dirigido
}

// Función auxiliar para DFS
void DFSUtil(int v, std::vector<bool>& visitado, const std::vector<std::list<int>>& listaAdyacencia)
{
```

```
        visitado[v] = true;
        std::cout << v << " ";

        for (auto i : listaAdyacencia[v]) {
                if (!visitado[i]) {
                        DFSUtil(i, visitado, listaAdyacencia);
                }
        }
}

// Función para realizar DFS desde un vértice específico

void DFS(int v, const std::vector<std::list<int>>& listaAdyacencia) {
        std::vector<bool> visitado(listaAdyacencia.size(), false);
        DFSUtil(v, visitado, listaAdyacencia);
        std::cout << std::endl;
}

// Función para realizar BFS desde un vértice específico

void BFS(int v, const std::vector<std::list<int>>& listaAdyacencia) {
        std::vector<bool> visitado(listaAdyacencia.size(), false);
        std::queue<int> cola;
        visitado[v] = true;
        cola.push(v);

        while (!cola.empty()) {
                v = cola.front();
                cola.pop();
                std::cout << v << " ";

                for (auto i : listaAdyacencia[v]) {
                        if (!visitado[i]) {
                                visitado[i] = true;
                                cola.push(i);
                        }
                }
        }
        std::cout << std::endl;
}

// Función para imprimir el grafo

void imprimirGrafo(const std::vector<std::list<int>>& listaAdyacencia) {
        for (size_t i = 0; i < listaAdyacencia.size(); ++i) {
                std::cout << "Nodo " << i << ":";
                for (auto j : listaAdyacencia[i]) {
                        std::cout << " -> " << j;
                }
                std::cout << std::endl;
        }
}

int main() {
        int numeroDeNodos = 5;
        std::vector<std::list<int>> listaAdyacencia(numeroDeNodos);

        // Insertar aristas
        insertarAristaLista(listaAdyacencia, 0, 1);
        insertarAristaLista(listaAdyacencia, 0, 4);
        insertarAristaLista(listaAdyacencia, 1, 2);
        insertarAristaLista(listaAdyacencia, 1, 3);
        insertarAristaLista(listaAdyacencia, 2, 3);
        insertarAristaLista(listaAdyacencia, 3, 4);

        std::cout << "Grafo:" << std::endl;
        imprimirGrafo(listaAdyacencia);

        std::cout << "DFS desde el nodo 0:" << std::endl;
```

```
    DFS(0, listaAdyacencia);

    std::cout << "BFS desde el nodo 0:" << std::endl;
    BFS(0, listaAdyacencia);

    std::cout << "Eliminar arista entre el nodo 1 y el nodo 2" << std::endl;
    eliminarAristaLista(listaAdyacencia, 1, 2);

    std::cout << "Grafo después de eliminar la arista:" << std::endl;
    imprimirGrafo(listaAdyacencia);

    return 0;
}
```

Explicación:

insertarAristaLista: Inserta una arista entre los nodos u y v. Si el grafo es no dirigido, la arista se agrega en ambas direcciones.

eliminarAristaLista: Elimina una arista entre los nodos u y v. Si el grafo es no dirigido, la eliminación se realiza en ambas direcciones.

DFSUtil y DFS: Realizan la búsqueda en profundidad. **DFSUtil** es una función auxiliar que realiza la búsqueda recursiva. DFS inicializa el proceso y controla el estado de los nodos visitados.

BFS: Realiza la búsqueda en anchura utilizando una cola para explorar todos los nodos a una distancia determinada antes de pasar a la siguiente.

imprimirGrafo: Imprime el grafo en formato de lista de adyacencia.

Este ejemplo cubre la creación de un grafo, la inserción y eliminación de aristas, y las búsquedas en profundidad y en anchura