OPERACIONES BASICAS EN UN GRAFO

FUENTES GUALOTUÑA JORGE LUIS
JACOME CALAHORRANO MICAELA VICTORIA
ROGERON MAILA MATEO NICOLAY

GRAFOS

Son estructuras de datos que se les utiliza para representar relaciones entre entidades

NODOS

Objetos o entidades del sistemaa

ARISTAS

Dependencias o relaciones entre los objetos

$$G = (V, E)$$

USOS

- Modela relaciones y estructuras en:
  - Redes
  - IA
  - Algoritmos
  - Optimización

DEFINICIONES FUNDAMENTALES

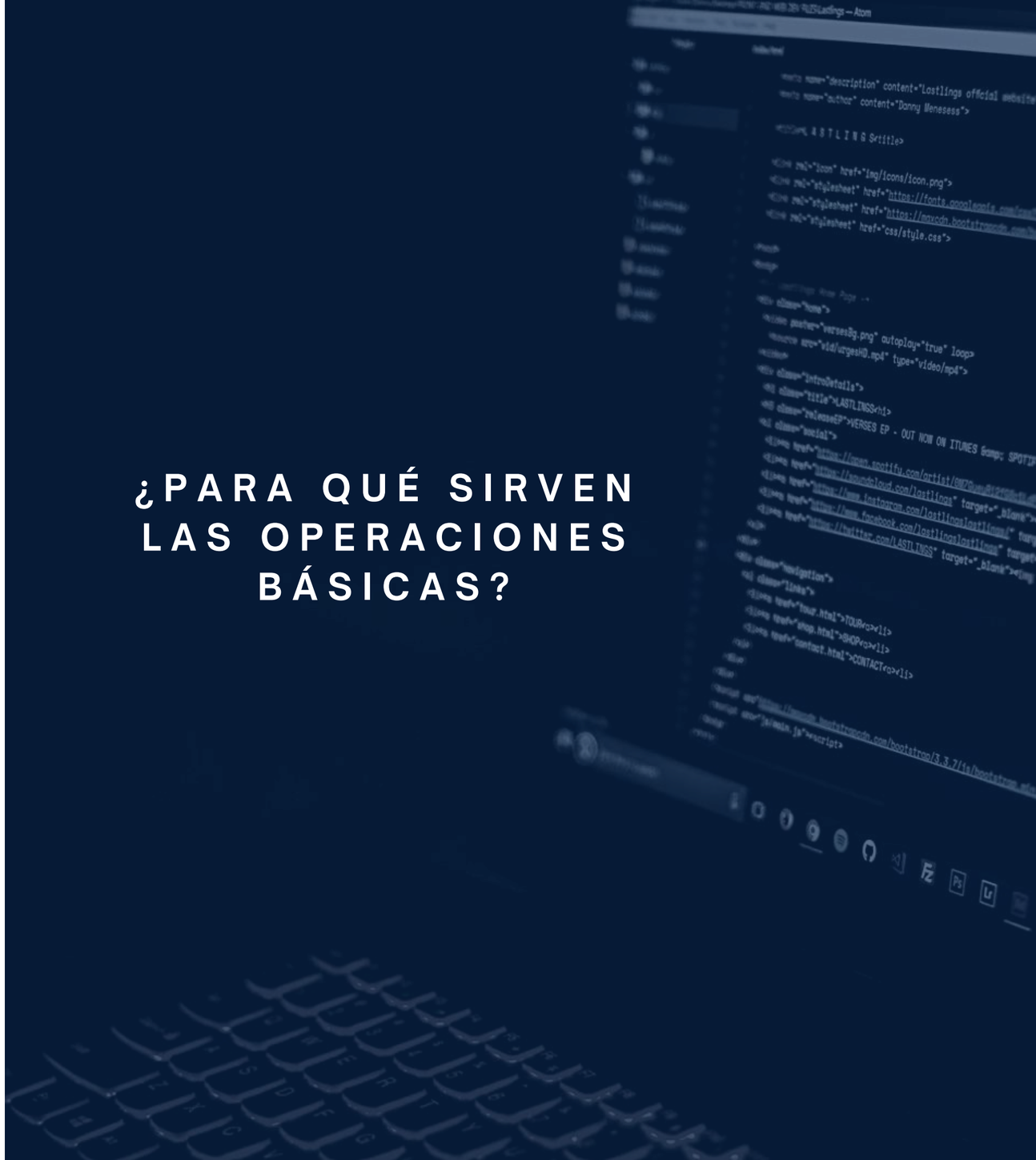
OPERACIONES FUNDAMENTALES

- Insertar nodo
- Insertar arista
- Eliminar nodo
- Eliminar arista
- Buscar nodos o aristas
- Recorridos: BFS y DFS
- Cálculo de propiedades:
  - Grado
  - Conectividad
  - Caminos mínimos

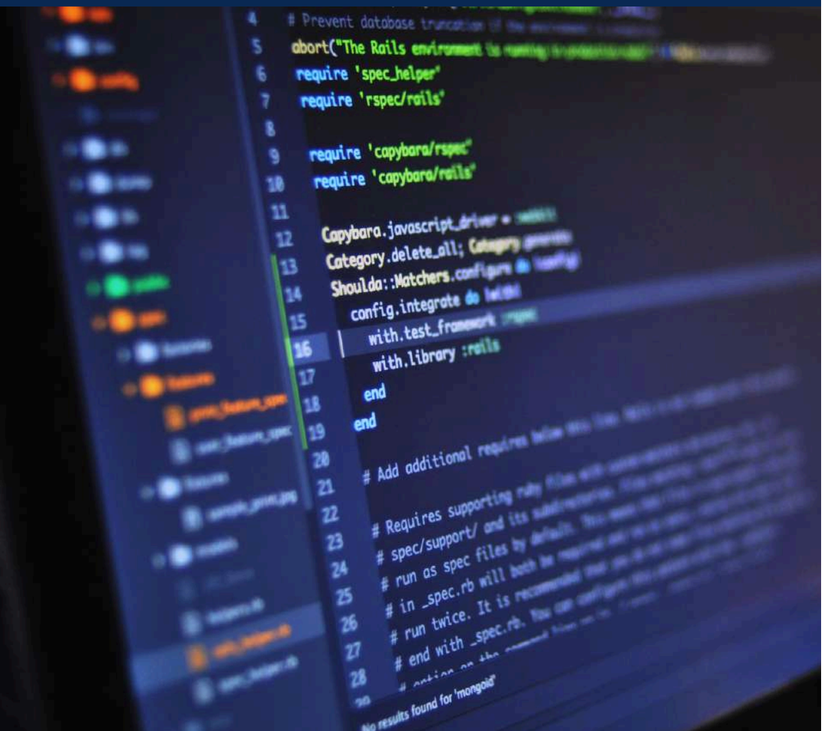
IMPORTANCIA

- Base para algoritmos como:
  - Dijkstra
  - Prim
  - Kruskal
- Usado en software que maneja relaciones complejas.

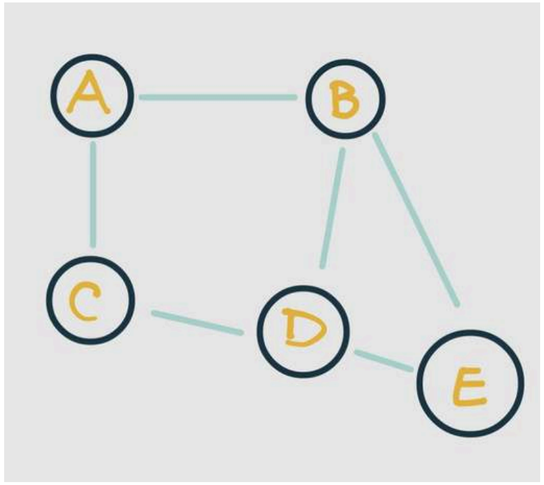
¿PARA QUÉ SIRVEN LAS OPERACIONES BÁSICAS?



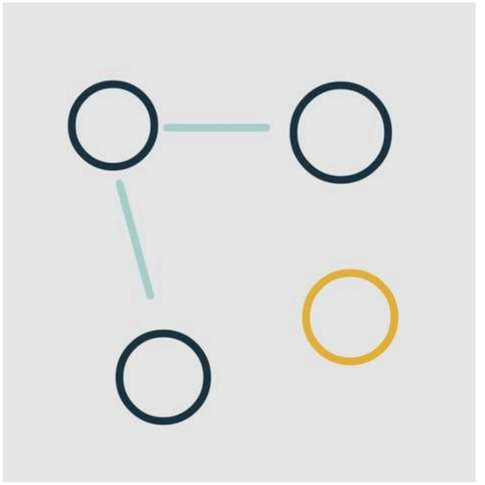
REPRESENTACIONES GRÁFICAS E IMÁGENES



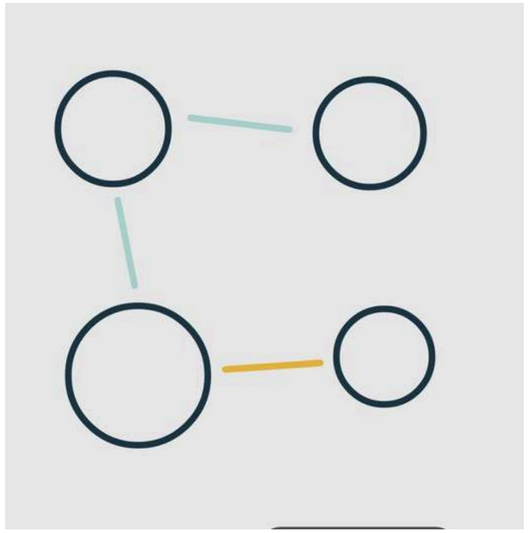
GRAFO



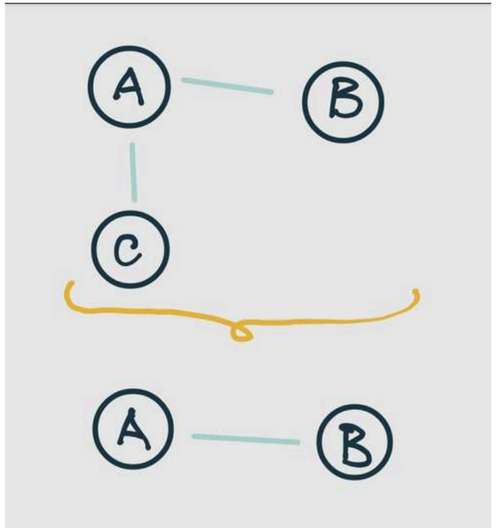
INSERCIÓN DE NODO



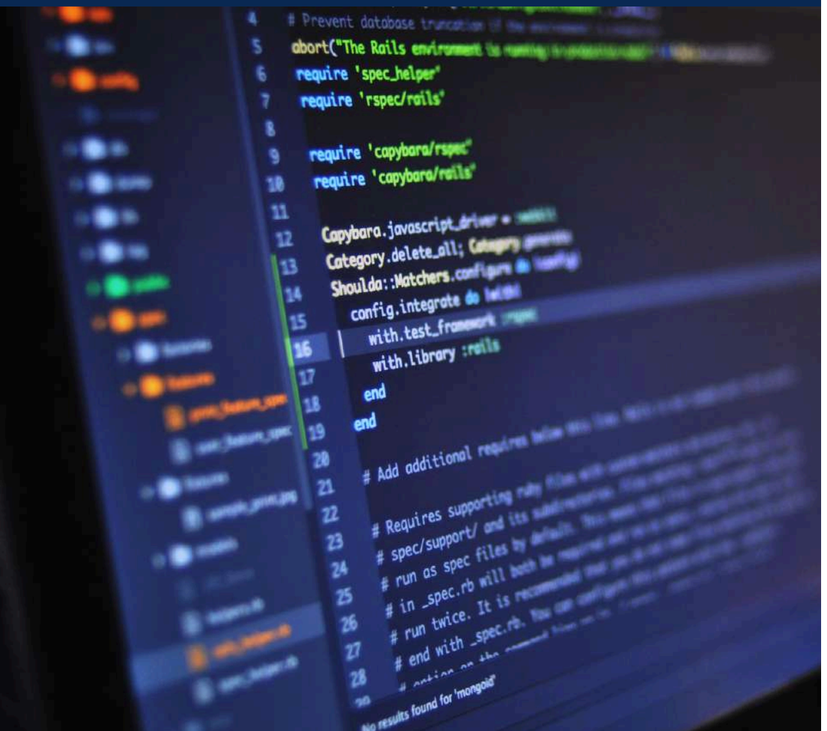
INSERCIÓN DE ARISTA



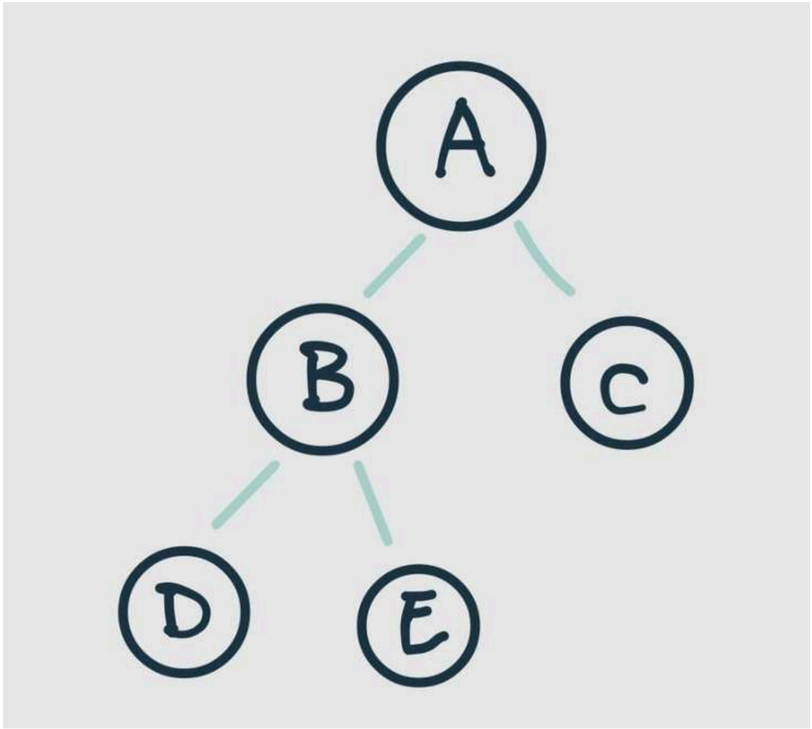
ELIMINACIÓN DE NODO



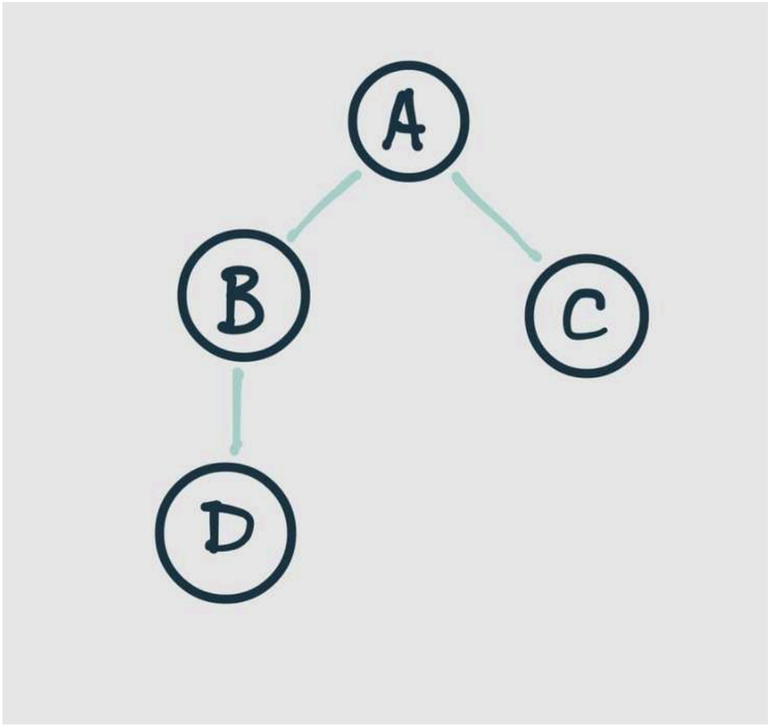
REPRESENTACIONES GRÁFICAS E IMÁGENES



BSF



DFS



SINTAXIS GENERAL

MATRIZ DE ADYACENCIA

```
const int N = 5;
int matriz[N][N] = {0};

// Insertar arista entre i y j
void insertarArista(int i, int j) {
    matriz[i][j] = 1;
    matriz[j][i] = 1; // si el grafo es no dirigido
}
```

LISTA DE ADYACENCIA

```
#include <iostream>
#include <vector>
using namespace std;

vector<vector<int>> grafo;

void insertarNodo(int n) {
    grafo.resize(n);
}

void insertarArista(int u, int v) {
    grafo[u].push_back(v);
    grafo[v].push_back(u); // si es no dirigido
}
```

SINTAXIS GENERAL DE OPERACIONES BASICAS

INSERTAR NODO

```
grafo.push_back(vector<int>());
```

INSERTAR ARISTA

```
grafo[u].push_back(v);
```

ELIMINAR ARISTA

```
grafo[u].erase(remove(grafo[u].begin(),
grafo[u].end(), v), grafo[u].end());
```

BFS

```
#include <queue>

void bfs(int inicio) {
    vector<bool> visitado(grafo.size(), false);
    queue<int> q;

    visitado[inicio] = true;
    q.push(inicio);

    while(!q.empty()) {
        int actual = q.front(); q.pop();
        cout << actual << " ";

        for(int vecino : grafo[actual]) {
            if(!visitado[vecino]) {
                visitado[vecino] = true;
                q.push(vecino);
            }
        }
    }
}
```

DFS

```
vector<bool> visitado;

void dfs(int nodo) {
    visitado[nodo] = true;
    cout << nodo << " ";

    for(int vecino : grafo[nodo]) {
        if(!visitado[vecino]) {
            dfs(vecino);
        }
    }
}
```

SINTAXIS ELIMINACIÓN DE NODO

LISTA DE ADYACENCIA

ELIMINACIÓN LOGICA

```
void eliminarNodo(int u) {
    // 1. Vaciar la lista del nodo u
    grafo[u].clear();

    // 2. Eliminar referencias a u en otros nodos
    for (auto &lista : grafo) {
        lista.erase(remove(lista.begin(), lista.end(), u),
lista.end());
    }
}
```

ELIMINACIÓN REAL

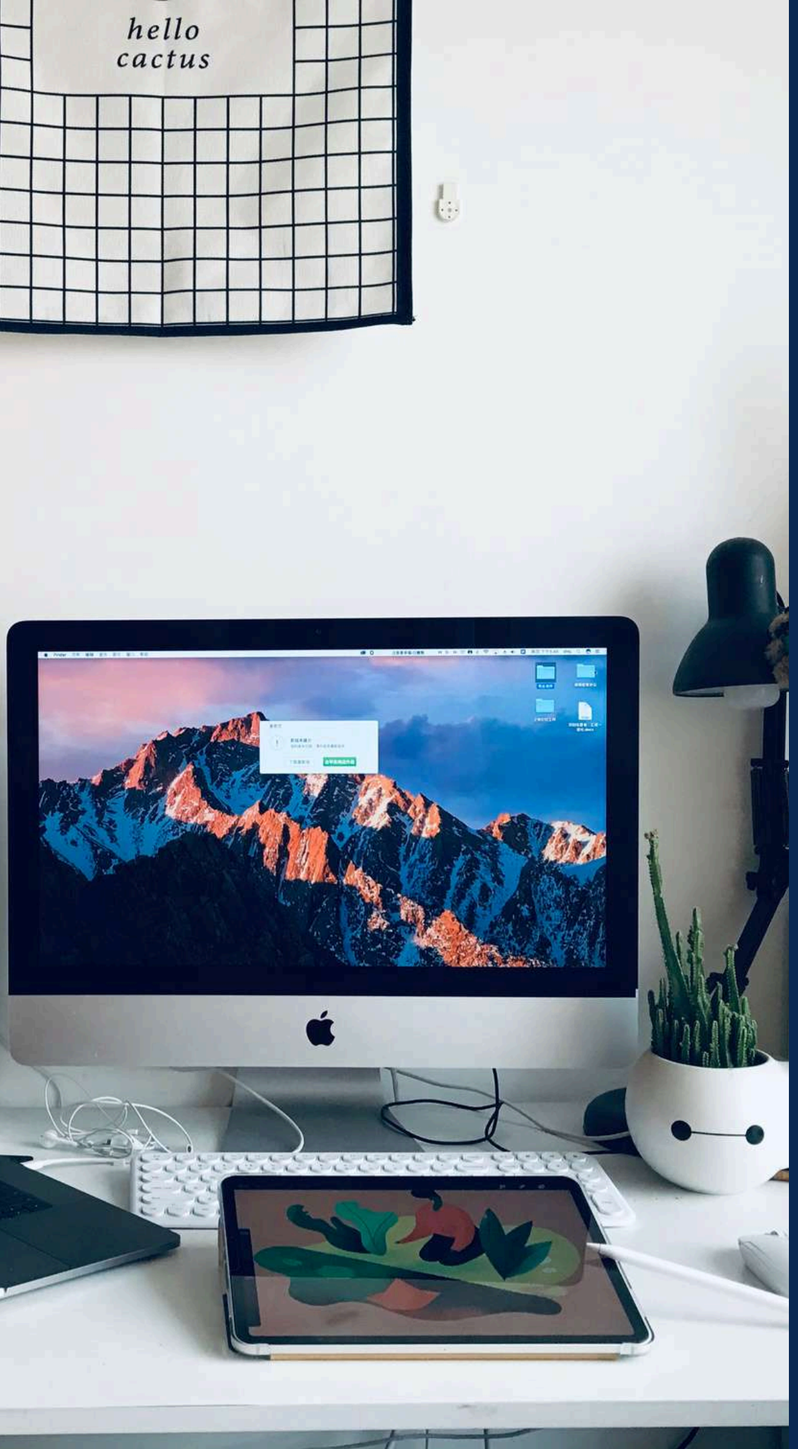
```
void eliminarNodoReal(int u) {
    // 1. Eliminar todas las aristas hacia u
    for (auto &lista : grafo) {
        lista.erase(remove(lista.begin(), lista.end(), u),
lista.end());
    }

    // 2. Eliminar completamente la lista del nodo u
    grafo.erase(grafo.begin() + u);

    // 3. Reajustar valores mayores a u
    for (auto &lista : grafo) {
        for (int &v : lista) {
            if (v > u) v--;   // bajar índices
        }
    }
}
```

MATRIZ DE ADYACENCIA

```
void eliminarNodoMatriz(int u) {
    for (int i = 0; i < N; i++) {
        grafo[u][i] = 0;  // eliminar fila
        grafo[i][u] = 0;  // eliminar columna
    }
}
```

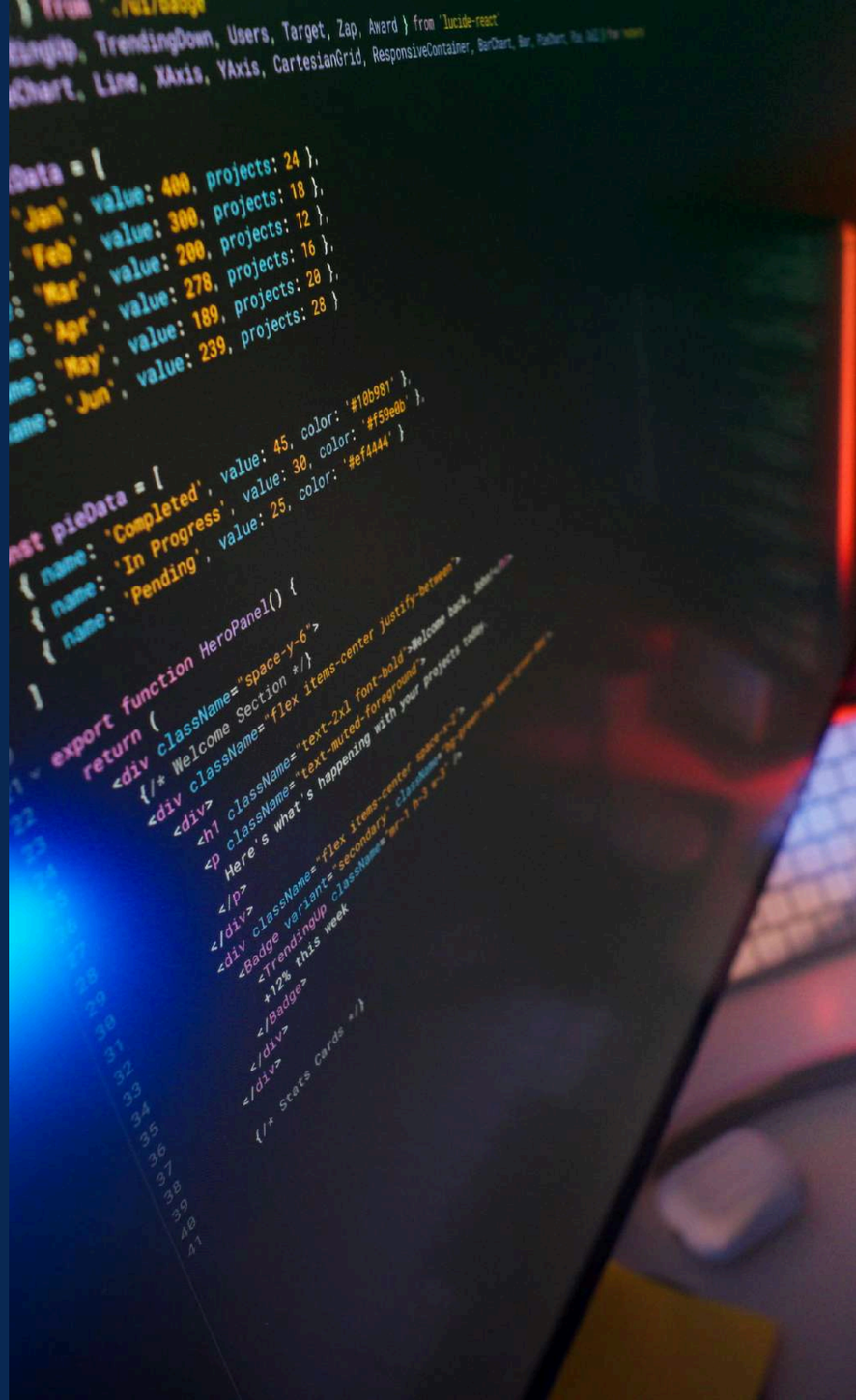


4/ VENTAJAS Y DESVENTAJAS

Estrategia de Búsqueda	Origen en la Tabla	Ventajas Fundamentales (Operaciones Básicas)	Desventajas Fundamentales (Límites de la Operación)
Búsqueda Básica (BFS/DFS)	Recorrido General (BFS/DFS)	Simplicidad y Eficiencia: Operación de visita y consulta de vecinos.	La limitación fundamental es que su operación de visita **ignora cualquier ponderación**. Si un grafo tiene pesos, la simple visita por capas no garantiza encontrar el camino más corto.
Relajación Voraz (Dijkstra)	Dijkstra	Eficiencia y Prioridad: La operación de consulta y relajación se optimiza con colas de prioridad (heap), siendo el método más rápido para el costo positivo.	Su desventaja no es la velocidad (es rápida), sino la **falta de robustez**. La operación de relajación confía ciegamente en la distancia más corta actual, por lo que un peso negativo rompe esa confianza.
Relajación Repetida (Bellman-Ford)	Bellman-Ford	Robustez Completa: La operación de relajación se repite.	Su desventaja es el **exceso de trabajo**. Su operación de relajación se repite forzosamente $\|V\|-1$ veces, siendo ineficiente cuando no hay pesos negativos que corregir.

5/ COMPARACIÓN

Característica	Algoritmo de Dijkstra	Algoritmo de Bellman-Ford
Problema que Resuelve	Camino más corto desde una única fuente (SSP).	Camino más corto desde una única fuente (SSP).
Pesos de Aristas	Solo acepta pesos no negativos (≥0).	Acepta pesos negativos.
Ciclos de Peso Negativo	No aplicable (el algoritmo fallaría de forma impredecible).	Detecta la existencia de un ciclo de peso negativo.
Complejidad Temporal	O	O(|V| * |E|)
Principio Operacional	Voraz (Greedy). Expande la distancia conocida más corta hasta un vértice y la establece como "permanente".	Programación Dinámica (Relajación). Repite el proceso de relajación de todas las aristas.
Aplicación Típica	Sistemas de navegación (GPS), enrutamiento de red (OSPF).	Enrutamiento de red donde los costos pueden ser variables (por ejemplo, en RIP), detección de arbitraje.



Código de ejemplo {

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <tuple>
#include <limits>
#include <algorithm>
#include <string>
#include <iomanip>

using namespace std;

// Definición de una arista: (Origen, Destino, Recompensa/Costo)
typedef tuple<string, string, double> MisionRelacion;
```

```
Función auxiliar para imprimir el estado actual de las recompensas (sin cambios)
d imprimir_recompensas(int iteracion, const map<string, double>& recompensas, const vector<string>& nodos) {
 cout << "\n=============================================================================================================\n";
 if (iteracion == 0) {
 |   cout << "ESTADO INICIAL (Iteracion 0)";
 } else {
 |   cout << "ESTADO DESPUÉS DE ITERACIÓN " << iteracion;
 }
 cout << "\n=============================================================================================================\n";

 // Encabezados de la tabla
 cout << "| " << setw(20) << left << "MISIÓN (NODO)" << " | " << setw(18) << right << "RECOMPENSA ACUMULADA" << " |" << end
 cout << "-------------------------------------------------------" << endl;

 // Cuerpo de la tabla
 for (const string& nodo : nodos) {
 |   double recompensa = recompensas.at(nodo);
 |   cout << "| " << setw(20) << left << nodo << " | ";
 |
 |   if (recompensa == -numeric_limits<double>::infinity()) {
 |   |   cout << setw(18) << right << "INF NEGATIVO";
 |   } else {
 |   |   cout << setw(18) << right << fixed << setprecision(1) << recompensa;
 |   }
 |   cout << " |" << endl;
 }
 cout << "=============================================================================================================\n";
```

```
// Implementación del Algoritmo de Bellman-Ford (adaptado para Máxima Recompensa)
pair<map<string, double>, bool> bellman_ford_recompensa(
    const vector<string>& todos_los_nodos,
    const vector<MisionRelacion>& relaciones,
    const string& inicio
) {
    // ... [Inicialización de estructuras] ...
    map<string, double> recompensas;
    map<string, string> predecesores;

    for (const string& nodo : todos_los_nodos) {
        recompensas[nodo] = -numeric_limits<double>::infinity();
        predecesores[nodo] = "-1";
    }
    recompensas[inicio] = 0.0;

    int V = todos_los_nodos.size();

    imprimir_recompensas(0, recompensas, todos_los_nodos);
```

```cpp
// 2. Relajación de Relaciones (V - 1) veces
for (int i = 0; i < V - 1; ++i) {
    bool cambiado = false;

    cout << "\n--- RELAJACIÓN DE ARISTAS EN ITERACIÓN " << i + 1 << " ---" << endl;
    bool hubo_mejora_en_iteracion = false; // Flag para saber si imprimir "NO HUBO MEJORAS"

    // Usamos un vector temporal para almacenar los detalles de los cambios
    vector<string> cambios_detalle_vertical;

    for (const auto& relacion : relaciones) {
        string u, v;
        double recompensa;
        tie(u, v, recompensa) = relacion;

        if (recompensas[u] != -numeric_limits<double>::infinity() && recompensas[u] + recompensa > recompensas[v]) {

            // Formato de salida vertical
            string detalle = "  -> RELAX: " + u + " -> " + v + " | Gana: " + to_string((int)recompensa) +
                             " | Antiguo: " + to_string((int)recompensas[v]) +
                             " | Nuevo Total: " + to_string((int)(recompensas[u] + recompensa));
            cambios_detalle_vertical.push_back(detalle);

            recompensas[v] = recompensas[u] + recompensa;
            predecesores[v] = u;
            cambiado = true;
            hubo_mejora_en_iteracion = true;
        }
    }
```

```
        if (hubo_mejora_en_iteracion) {
            cout << "CAMBIOS REALIZADOS EN ESTA PASADA:\n";
            for (const string& detalle : cambios_detalle_vertical) {
                cout << detalle << endl;
            }
        } else {
            cout << "NO HUBO MEJORAS EN ESTA ITERACIÓN." << endl;
        }

        imprimir_recompensas(i + 1, recompensas, todos_los_nodos);

        if (!cambiado) {
            cout << "\n[FIN TEMPRANO]: El grafo ha convergido después de la Iteración " << i << ".\n" << endl;
            break;
        }
    }
```

```
    // 3. Chequeo de Ciclos de Recompensa Positiva (V-ésima Iteración)
    cout << "\n--- PRUEBA DE CICLO DE GRINDEO POSITIVO (Iteracion V = " << V << ") ---" << endl;
    bool ciclo_infinito = false;

    for (const auto& relacion : relaciones) {
        string u, v;
        double recompensa;
        tie(u, v, recompensa) = relacion;

        if (recompensas[u] != -numeric_limits<double>::infinity() && recompensas[u] + recompensa > recompensas[v]) {
            // Identificar la arista específica que permite el ciclo!
            cout << "  ¡CICLO DETECTADO! La arista " << u << "->" << v
                 << " permite otra mejora (" << recompensas[u] + recompensa << " > " << recompensas[v] << ").\n";
            ciclo_infinito = true;
            break;
        }
    }

    return {recompensas, ciclo_infinito};
}
```

```cpp
// Función principal de ejemplo (Main)
int main() {
    // --- Modelado de Misiones (Nodos) ---
    vector<string> misiones_nodos = {
        "Inicio", "MS_1: Tutorial", "MS_2: Recoleccion",
        "MP_1: Explorar", "MP_2: Asalto", "Jefe", "Cofre_Secreto"
    };

    // --- Definición de las Relaciones (Aristas y Pesos) ---
    vector<MisionRelacion> relaciones_misiones = {
        {"Inicio", "MS_1: Tutorial", 100.0},
        {"Inicio", "MP_1: Explorar", 50.0},
        {"MS_1: Tutorial", "MS_2: Recoleccion", 200.0},

        // Bucle de grindeo (recompensa neta positiva: 500 + 10 = 510)
        {"MS_2: Recoleccion", "MP_2: Asalto", 500.0},
        {"MP_2: Asalto", "MS_2: Recoleccion", 10.0},

        {"MP_1: Explorar", "MS_2: Recoleccion", 150.0},
        {"MP_2: Asalto", "Jefe", 800.0},
        {"Jefe", "Cofre_Secreto", 1000.0}
    };

    string nodo_inicial = "Inicio";
    string nodo_destino = "Cofre_Secreto";

    // Ejecutar el algoritmo
    auto resultado = bellman_ford_recompensa(misiones_nodos, relaciones_misiones, nodo_inicial);
    map<string, double> recompensas = resultado.first;
    bool ciclo_infinito = resultado.second;
```

```cpp
    // Imprimir los resultados
    cout << "\n================================================================\n";
    cout << "--- ANALISIS FINAL: RUTA DE MAXIMA RECOMPENSA ---\n";
    cout << "Objetivo: " << nodo_destino << endl;

    if (ciclo_infinito) {
        cout << "\n****************************************************************\n";
        cout << "ADVERTENCIA: Se detecto un BUCLE DE GRINDEO POSITIVO (Recompensa Infinita).\n";
        cout << "El camino de maxima recompensa es teoricamente infinito.\n";
        cout << "****************************************************************\n";
    }

    cout << "\nRecompensa maxima alcanzada en el destino: ";
    if (recompensas.count(nodo_destino) && recompensas.at(nodo_destino) == -numeric_limits<double>::infinity()) {
        cout << "No alcanzable." << endl;
    } else {
        // Mostrar la recompensa máxima
        cout << recompensas.at(nodo_destino) << " XP/Oro" << (ciclo_infinito ? " (El valor puede ser arbitrariamente alto)" : "") << endl;
    }

    return 0;
}
```

}

APLICACIÓN PRÁCTICA

APLICACIÓN PRÁCTICA

Plataformas como Facebook, LinkedIn o Instagram utilizan grafos para modelar sus redes sociales. Cada usuario es un vértice y cada conexión de amistad/seguimiento es una arista.

Búsqueda en anchura (BFS): Para encontrar "amigos de amigos" y sugerir nuevas conexiones

Búsqueda en profundidad (DFS): Para explorar comunidades o grupos dentro de la red

Detección de componentes conexos: Identificar usuarios aislados o comunidades separadas

Cálculo del grado: Determinar usuarios influyentes (con muchas conexiones)

- Representan naturalmente relaciones entre entidades
- Permiten encontrar caminos más cortos entre usuarios
- Facilitan el análisis de comunidades (clustering)
- Optimizan la difusión de contenido a través de la red
- Las operaciones básicas (recorridos, búsqueda) son la base para algoritmos más complejos de recomendación

APLICACIÓN PRÁCTICA

CONCLUSIONES

- Los grafos son estructuras fundamentales con vértices y aristas
- Las operaciones básicas incluyen: inserción/eliminación de vértices/aristas, recorridos (BFS/DFS), búsqueda de caminos
- La representación (matriz de adyacencia o lista de adyacencia) afecta la eficiencia de las operaciones
- BFS es óptimo para caminos más cortos en grafos no ponderados
- DFS es útil para topologías, componentes conexos y detectar ciclos

Desafíos y Dificultades

- Complejidad algorítmica: Elegir entre representaciones según la densidad del grafo
- Algoritmos de recorrido: Diferenciar casos de uso entre BFS y DFS según el problema
- Gestión de memoria: En grafos muy grandes y dispersos, las matrices de adyacencia son ineficientes
- Detección de ciclos: Implementar correctamente DFS para identificar ciclos en grafos dirigidos y no dirigidos
- Casos especiales: Manejar grafos desconexos, dirigidos vs no dirigidos, ponderados vs no ponderados


MUCHAS GRACIAS