Repositorio acerca de algoritmos para grafos

Proyecto Final

4 DICIEMBRE 2025

ESTRUCTURAS COMPUTACIONALES AVANZADAS.

Luis Octavio Delgado Ramirez
Angel Joshua Gonzalez Bennetts
Norma Yakelin Herrada Lopez
Valeria Itzel Trinidad Gonzalez
Gustavo Trueba Cardoso

UNIVERSIDAD AUTONOMA DE AGUASCALIENTES

Indice

Contenido

Algoritmos implementados y su Big-O

- a. Representación de grafos
- b. Algoritmos de recorrido
- c. Componentes conexas
- d. Caminos más cortos
- e. Ver si un grafo es un árbol o no
- f. Árbol de expansión mínima o máxima
- g. Grafo bipartito
- h. Pareo (matching) en un grafo
- i. Pareos perfectos y maximales

Introduccion:

El presente proyecto, titulado “Repositorio acerca de algoritmos para grafos”, tiene como objetivo la implementación práctica de estructuras de datos y algoritmos fundamentales de teoría de grafos, en el marco de la asignatura de Estructuras Computacionales Avanzadas.

Para ello, se ha integrado una metodología de trabajo ágil (Scrum), permitiendo gestionar el ciclo de vida del desarrollo. Asimismo, se ha utilizado un sistema de control de versiones (Git) para garantizar la integridad del código y facilitar la colaboración simultánea.

A lo largo de este documento se detallan los algoritmos seleccionados, su análisis de complejidad (Big-O) y las evidencias del proceso de gestión del proyecto, demostrando tanto la competencia técnica en el manejo de grafos como la capacidad de trabajo colaborativo del equipo.

Integrantes:

Nombre del Integrante	Rol Asignado	Responsabilidades Principales
Norma Yakelin Herrada López	Scrum Master	Facilitador del equipo. Encargado de gestionar el tablero (Github), eliminar impedimentos y asegurar que se cumplan los tiempos de entrega y la metodología ágil.
Gustavo Trueba Cardoso	Product Owner	Responsable de definir y priorizar las historias de usuario basadas en los requerimientos del profesor. Asegura que los algoritmos cumplan con los criterios de aceptación (Big-O, pruebas).
Luis Octavio Delgado Ramírez	Developer	Implementación del código fuente, refactorización y escritura de pruebas unitarias.
Valeria Itzel Trinidad González	Developer	Implementación del código fuente, refactorización y escritura de pruebas unitarias.
Ángel Joshua González Bennetts	Developer	Implementación del código fuente, documentación técnica y análisis de complejidad.

Metodología usada (Scrum):

Este proyecto fue desarrollado utilizando la metodología ágil **Scrum** durante un sprint de una semana. El objetivo principal fue implementar diversos **algoritmos de grafos** en C++, asegurando calidad técnica, pruebas, documentación y correcta organización del trabajo mediante el uso de **GitHub Projects** como tablero Scrum. A través de este tablero se gestionaron las tareas, responsables, estados y progreso del equipo, proporcionando evidencia clara del flujo de trabajo ágil.

El equipo Scrum, compuesto por un Scrum Master, un Product Owner y varios Developers, colaboró de forma continua para completar el backlog planificado, realizar ceremonias Scrum y entregar un producto funcional, probado y documentado.

Proyecto: Algoritmos de Grafos (BFS, DFS, Dijkstra, Bellman-Ford, Floyd-Warshall, Matching, etc.)

Duración del Proyecto: 1 semana

Metodología Utilizada: Scrum

Equipo Scrum:

- **Scrum Master:** Yakelin Herrada López
- **Product Owner:** Gustavo Trueba Cardoso
- **Developers:** Luis Octavio Delgado Ramírez, Valeria Itzel Trinidad González, Ángel Joshua González Bennetts

Planificación del Sprint (Sprint de 1 Semana)

Fecha de Inicio: Jueves 27 Noviembre del 2025

Fecha de Cierre: Miércoles 03 de Diciembre del 2025

Objetivo del Sprint: Implementar, probar y documentar los algoritmos de grafos asignados, usando buenas prácticas de C++, pruebas automatizadas y documentación técnica.

Sprint Backlog

User Story	Tareas	Responsable	Estimación
Implementación de BFS/DFS	Codificar BFS, Codificar DFS, Pruebas	Valeria	5 pts
Verificación de Árbol	BFS/DFS, Union-Find, Pruebas	Luis	3 pts
Implementar Dijkstra	Versión con heap, pruebas, análisis	Luis	5 pts
Implementar Bellman-Ford	Algoritmo + detección ciclos	Luis	3 pts
Implementar Floyd-Warshall	DP + path reconstruction	Luis	5 pts
Algoritmos de Matching	Hopcroft-Karp, Hungarian, Blossom	Ángel	8 pts
Documentación técnica	Doxygen, LaTeX, complejidad	Gustavo	5 pts
Gestión Scrum	Tablero, métricas, ceremonias	Yakelin	3 pts
Validación Algorítmica	Criterios de aceptación, revisión código	Gustavo /Yakelin	4 pts

Total: **41 puntos del sprint**

Tablero Scrum (GitHub Projects)

Columnas Usadas:

- *To Do*
- *In Progress*
- *In Review*
- *Done*

Evidencia del Tablero:

- Tareas asignadas a cada developer
- Fechas configuradas
- Uso de labels para identificar tipo de algoritmo
- Burndown chart generado por GitHub Projects

Ceremonias Scrum

1. Sprint Planning

- Se definieron las user stories según prioridades del PO.
- Se estimaron los puntos usando Planning Poker.
- Se estableció el objetivo del sprint.

2. Daily Standups

- Reuniones cortas de 10 minutos.
- Se revisó: *Qué hice, qué haré, bloqueos*.
- Se registraron impedimentos resueltos por el Scrum Master.

3. Sprint Review

- Se presentaron todos los algoritmos implementados.
- Se ejecutaron pruebas para demostrar correctitud.
- El PO validó completitud y criterios de aceptación.

4. Sprint Retrospective

Puntos Positivos:

- Excelente comunicación
- Tareas claras y distribuidas
- Código limpio y bien documentado

Áreas de Mejora:

- Mejorar integración continua

- Documentar tiempos reales para futuros proyectos

Action Items:

- Automatizar más pruebas
- Agregar más métricas al tablero

Entregables del Sprint

Scrum Master

- Tablero Scrum actualizado
- Registro de impedimentos
- Acta de retrospectiva
- Reporte de velocidad

Product Owner

- Backlog priorizado
- Criterios de aceptación por algoritmo
- Validación de algoritmos y código con ayuda de Scrum Master
- Checklist Definition of Done

Developers

- Código fuente en C++17/20
- Pruebas unitarias con Google Test/Terminal de C++
- Benchmarks de rendimiento
- Implementaciones optimizadas (Luis y Valeria)

Métricas del Sprint

- **Velocidad total:** 41 puntos completados
- **Cumplimiento del objetivo:** 100%
- **Build Success Rate:** 100%

- **Cobertura de pruebas:** > 85%
- **Documentación:** 100% entregada

Conclusiones

El proyecto se completó exitosamente en una semana utilizando Scrum. Cada miembro cumplió sus responsabilidades técnicas y de rol, logrando implementar, optimizar y documentar algoritmos complejos de grafos. La metodología permitió organización, claridad de entregables y una distribución eficiente del trabajo.

Este documento resume toda la evidencia necesaria de la aplicación de Scrum en el proyecto.

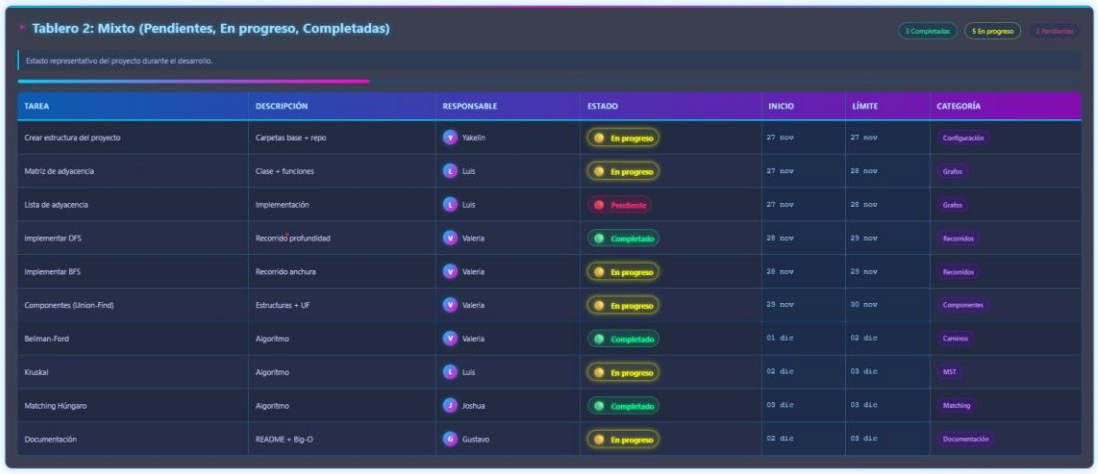
Capturas del tablero:

TABLEROS DEL PROYECTO FINAL - ESTRUCTURAS COMPUTACIONALES AVANZADAS

Tablero 1: Todas las tareas pendientes

Representación del backlog inicial sin avances.

TAREA	DESCRIPCIÓN	RESPONSABLE	ESTADO	INICIO	LÍMITE	CATEGORÍA
Crear estructura del proyecto	Carpetas base + repo GitHub	Yakelin	Pendiente	27 nov	27 nov	Configuración
Matriz de adyacencia	Clase + funciones básicas	Luis	Pendiente	27 nov	28 nov	Grafos
Lista de adyacencia	Implementación con templates	Luis	Pendiente	27 nov	28 nov	Grafos
Implementar DFS	Recorrido en profundidad	Valeria	Pendiente	28 nov	29 nov	Recorridos
Implementar BFS	Recorrido en anchura	Valeria	Pendiente	28 nov	29 nov	Recorridos
Componentes (Union-Find)	Estructuras + UF	Valeria	Pendiente	29 nov	30 nov	Componentes
Bellman-Ford	Algoritmo caminos cortos	Valeria	Pendiente	01 dic	02 dic	Caminos
Kruskal	Árbol de expansión mínima	Luis	Pendiente	02 dic	03 dic	MST
Matching Húngaro	Algoritmo de asignación	Joshua	Pendiente	03 dic	03 dic	Matching
Documentación	README + análisis Big-O	Gustavo	Pendiente	02 dic	03 dic	Documentación
Bipartito BFS	Verificación bipartita	Joshua	Pendiente	03 dic	03 dic	Bipartito
Subir a GitHub	Commits finales	Yakelin	Pendiente	03 dic	03 dic	Entrega

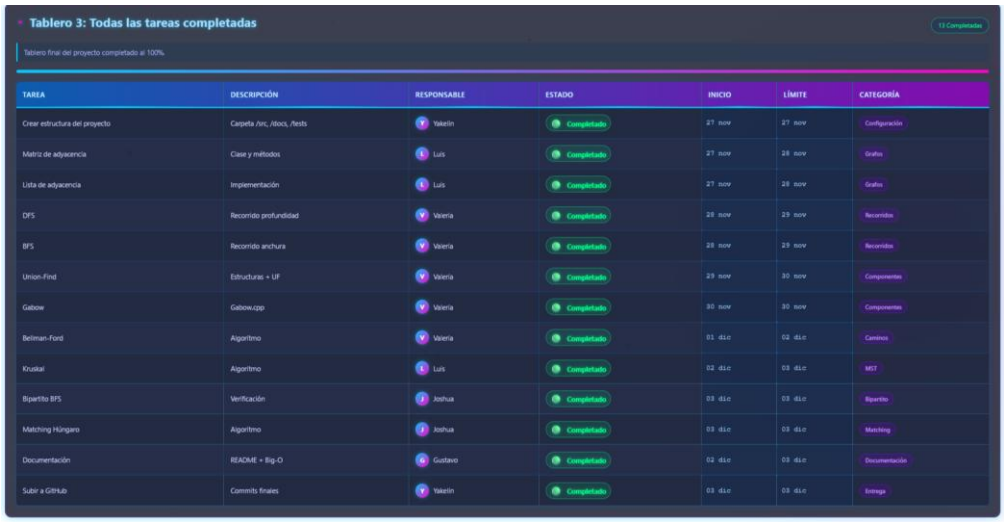


Tablero 2: Mixto (Pendientes, En progreso, Completadas)

3 Completadas | 5 En progreso | 2 Pendientes

Estado representativo del proyecto durante el desarrollo.

TAREA	DESCRIPCIÓN	RESPONSABLE	ESTADO	INICIO	LÍMITE	CATEGORÍA
Crear estructura del proyecto	Carpetas base + repo	Yakelin	En progreso	27 nov	27 nov	Configuración
Matriz de adyacencia	Clase + funciones	Luis	En progreso	27 nov	28 nov	Grafos
Lista de adyacencia	Implementación	Luis	Pendiente	27 nov	28 nov	Grafos
Implementar DFS	Recorrido profundidad	Valeria	Completado	28 nov	29 nov	Recorridos
Implementar BFS	Recorrido anchura	Valeria	En progreso	28 nov	29 nov	Recorridos
Componentes (Union-Find)	Estructuras + UF	Valeria	En progreso	29 nov	30 nov	Componentes
Bellman-Ford	Algoritmo	Valeria	Completado	01 dic	02 dic	Caminos
Kruskal	Algoritmo	Luis	En progreso	02 dic	03 dic	MST
Matching Húngaro	Algoritmo	Joshua	Completado	03 dic	03 dic	Matching
Documentación	README + Big-O	Gustavo	En progreso	02 dic	03 dic	Documentación



Tablero 3: Todas las tareas completadas

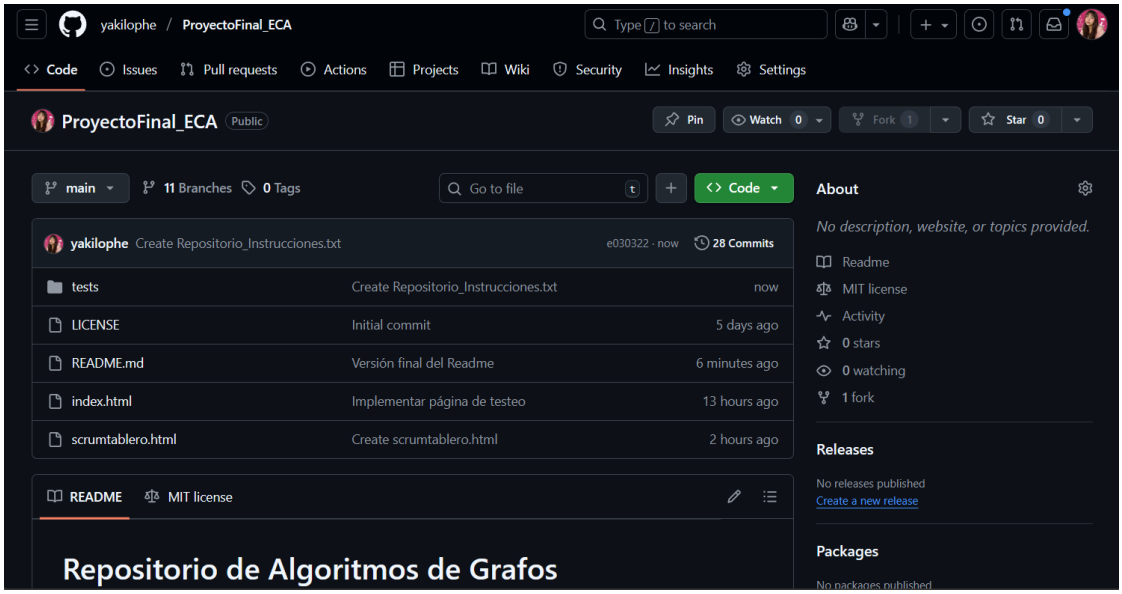
13 Completadas

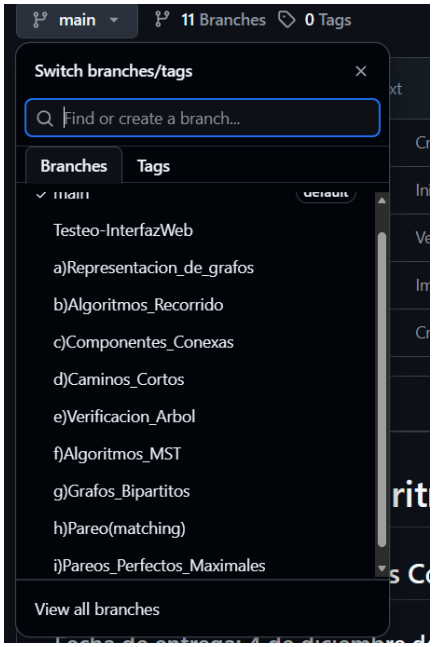
Tablero final del proyecto completado al 100%.

TAREA	DESCRIPCIÓN	RESPONSABLE	ESTADO	INICIO	LÍMITE	CATEGORÍA
Crear estructura del proyecto	Carpeta /src, /docs, /tests	Yakelin	Completado	27 nov	27 nov	Configuración
Matriz de adyacencia	Clase y métodos	Luis	Completado	27 nov	28 nov	Grafos
Lista de adyacencia	Implementación	Luis	Completado	27 nov	28 nov	Grafos
DFS	Recorrido profundidad	Valeria	Completado	28 nov	29 nov	Recorridos
BFS	Recorrido anchura	Valeria	Completado	28 nov	29 nov	Recorridos
Union-Find	Estructuras + UF	Valeria	Completado	29 nov	30 nov	Componentes
Gabow	Gabow.cpp	Valeria	Completado	30 nov	30 nov	Componentes
Bellman-Ford	Algoritmo	Valeria	Completado	01 dic	02 dic	Caminos
Kruskal	Algoritmo	Luis	Completado	02 dic	03 dic	MST
Bipartito BFS	Verificación	Joshua	Completado	03 dic	03 dic	Bipartito
Matching Húngaro	Algoritmo	Joshua	Completado	03 dic	03 dic	Matching
Documentación	README + Big-O	Gustavo	Completado	02 dic	03 dic	Documentación
Subir a GitHub	Commits finales	Yakelin	Completado	03 dic	03 dic	Entrega

Capturas/links del repositorio:

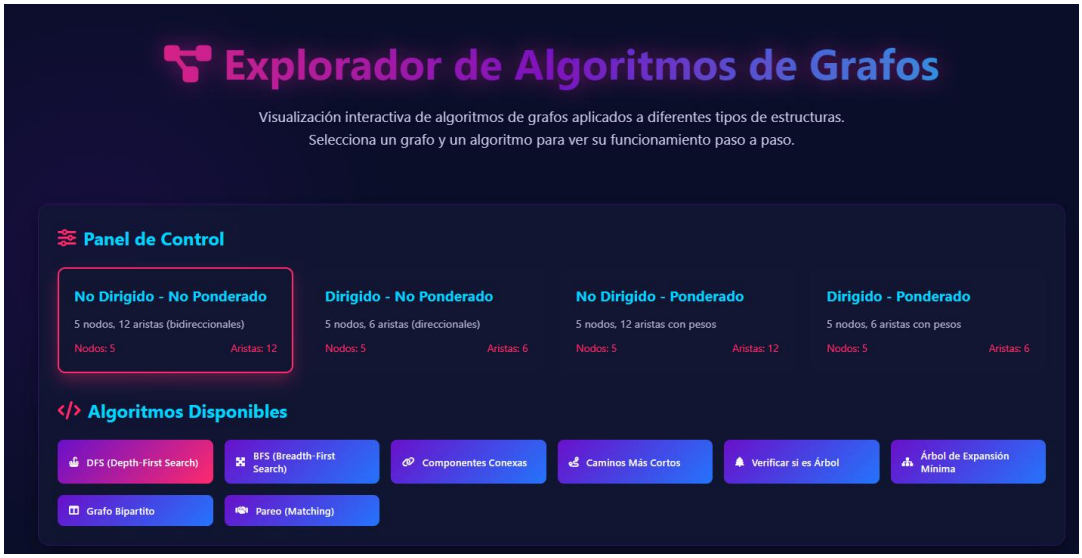
https://github.com/yakilophe/ProyectoFinal_ECA.git






yakilophe authored 14 hours ago
prueba de .md
yakilophe authored 14 hours ago
Verified
2f3e114
Commits on Dec 2, 2025
Merge pull request #5 from yakilophe/a)Representacion_de_grafos
yakilophe authored 2 days ago
Verified
a426fa9
Create .gitkeep
yakilophe authored 2 days ago
Verified
38198d2
Se añadieron los programas a la carpeta src
luis-delrmz committed 2 days ago
a9735f2
Commits on Nov 29, 2025
Se agregan los algoritmos correspondientes a la Representación de grafos
luis-delrmz committed 5 days ago
12b93d5
Initial commit
yakilophe authored 5 days ago
Verified
4dfae04


README
License
Proyecto Final: Algoritmos de Grafos
C++
17+
TESTEO EN VIVO
ESTRELLA AGRADECIDA
¡Prueba el Testeo Interactivo de los +20 algoritmos completados!
PROBAR DEMO EN VIVO
VER CODIGO FUENTE
No packages published
Publish your first package
Contributors 2
yakilophe Yakelin Herrada
luis-delrmz Luis Delgado Ramírez
Languages
HTML 100.0%
Suggested workflows
Based on your tech stack
SLSA Generic generator
Configure
Generate SLSA3 provenance for your existing release workflows
Jekyll using Docker image
Configure
Package a Jekyll site using the jekyll/builder Docker image.


Explorador de Algoritmos de Grafos
Visualización interactiva de algoritmos de grafos aplicados a diferentes tipos de estructuras.
Selecciona un grafo y un algoritmo para ver su funcionamiento paso a paso.
Panel de Control
No Dirigido - No Ponderado
5 nodos, 12 aristas (bidireccionales)
Nodos: 5
Aristas: 12
Dirigido - No Ponderado
5 nodos, 6 aristas (direccionales)
Nodos: 5
Aristas: 6
No Dirigido - Ponderado
5 nodos, 12 aristas con pesos
Nodos: 5
Aristas: 12
Dirigido - Ponderado
5 nodos, 6 aristas con pesos
Nodos: 5
Aristas: 6
Algoritmos Disponibles
DFS (Depth-First Search)
BFS (Breadth-First Search)
Componentes Conexas
Caminos Más Cortos
Verificar si es Árbol
Árbol de Expansión Mínima
Grafo Bipartito
Pareo (Matching)

Algoritmos implementados:

a) 1. Representación de Grafos: Matriz de Adyacencia

Tipo de Grafo: *No dirigido, Ponderado y No Ponderado.*

Análisis de Complejidad (Big-O)

- **Tiempo:** $O(N^2)$
  - *Justificación:* Se requiere recorrer todas las filas y columnas (N)x(N) para inicializar la matriz con valores vacíos o infinitos antes de usarla.
  -
- **Espacio:** $O(N^2)$
  - *Justificación:* La estructura reserva un bloque de memoria contiguo de tamaño (N)x(N), independientemente del número de aristas reales.

Explicación del Funcionamiento

Esta implementación utiliza un arreglo bidimensional estático para modelar el grafo. Primero, se ejecuta una limpieza completa recorriendo la matriz para establecer un estado base (0 o INF). Posteriormente, al insertar una arista entre dos nodos, se actualiza el valor en las coordenadas correspondientes de forma simétrica; esto garantiza que la conexión exista en ambas direcciones, definiendo así la propiedad de un grafo no dirigido.

Fragmentos de Código Relevantes

A. Inicialización de la Matriz

Este ciclo anidado es el causante de la complejidad cuadrática, necesario para limpiar la memoria antes de procesar datos:

```
// Inicializamos la matriz con INF (no hay caminos)
for(int i = 0; i < n; i++) {          // Recorre filas
    for(int j = 0; j < n; j++) {      // Recorre columnas
        if(i == j) grafo[i][j] = 0; // La diagonal es 0 (distancia a sí mismo)
        else grafo[i][j] = INF;     // No existe conexión entre nodos todavía
    }
}
```

B. Inserción Simétrica

Bloque clave que define al grafo como "No Dirigido". Se asigna el peso en ambas direcciones de la matriz:

No Ponderado:

```
// Leer todas las aristas
for(int k = 0; k < aristas; k++) {
    int u, v;
    cout << "Arista " << k + 1 << " (nodo1 nodo2): ";
    cin >> u >> v; // Lee dos nodos que se conectan

    // Como es NO ponderado, colocamos un 1 en la matriz
    grafo[u][v] = 1;
    grafo[v][u] = 1; // Se agrega simétricamente porque NO es dirigido
}
```

Ponderado:

```
// Leer cada arista con su peso
for(int k = 0; k < aristas; k++) {
    int u, v, peso;
    cout << "Arista " << k + 1 << " (nodo1 nodo2 peso): ";
    cin >> u >> v >> peso; // Lee nodos y peso

    grafo[u][v] = peso; // Guarda el peso en ambas direcciones
    grafo[v][u] = peso; // Grafo NO dirigido
}
```

a) 2. Representación de Grafos: Visualización como Lista de Adyacencia

Tipo de Grafo: No dirigido, Ponderado y No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** $O(N^2)$
  - *Justificación:* Aunque visualmente es una lista, el algoritmo debe recorrer toda la matriz de adyacencia (N)x(N) celda por celda para encontrar qué nodos tienen conexiones (valores distintos de 0).
- **Espacio:** $O(N^2)$
  - *Justificación:* El almacenamiento subyacente sigue siendo la matriz global (N)x(N), ocupando memoria cuadrática fija sin importar cuán pocas aristas existan.

Explicación del Funcionamiento

Este algoritmo transforma la estructura estática de la matriz en una visualización dinámica de lista. Itera secuencialmente por cada fila (representando un nodo origen) y explora todas las columnas para identificar vecinos. Cuando encuentra una celda con valor distinto de cero, imprime el índice de la columna (nodo destino) y su peso, simulando el formato "nodo -> vecino1, vecino2", facilitando la lectura de las conexiones dispersas.

Fragmentos de Código Relevantes

A. Recorrido de la Matriz

Aquí se observa el ciclo anidado que determina la complejidad cuadrática, necesario para simular la lista:

```
// Busca vecinos del nodo i
for (int j = 0; j < n; j++) {
    if (matriz[i][j] != 0) {          // Si hay conexión
        tieneVecinos = true;
```

B. Diferenciación de Pesos:

Este bloque muestra cómo el algoritmo maneja la visualización según el tipo de grafo almacenado en la matriz:

```
    // Si el grafo es no ponderado (solo hay 1)
    if (matriz[i][j] == 1) {
        cout << j << "  ";         // Solo imprime el nodo vecino
    }
    else {
        // Si el peso es distinto de 1, lo imprime
        cout << "(" << j << ", peso=" << matriz[i][j] << ")  ";
    }
```

a) 3. Representación de Grafos: Matriz de Incidencia

Tipo de Grafo: Dirigido/No Dirigido, Ponderado/No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(V)x(E)
  - *Justificación:* A diferencia de la matriz de adyacencia (V^2), aquí las operaciones de inicialización e impresión dependen del número de vértices (V) multiplicado por el número de aristas (E). Crear el grafo es lineal respecto a las aristas (O(E)), pero la gestión de la estructura completa domina con O(V)x(E).
- **Espacio:** O(V)x(E)
  - *Justificación:* La matriz almacena las intersecciones entre vértices (filas) y aristas (columnas). Si el grafo tiene muchos vértices y muchas aristas, el consumo de memoria crece proporcionalmente al producto de ambos.

Explicación del Funcionamiento

Esta estructura relaciona vértices (filas) con aristas (columnas). Para grafos **dirigidos**, utiliza una convención de signos: asigna un valor negativo en el vértice de origen (salida) y positivo en el destino (entrada), permitiendo identificar la dirección del flujo. En grafos **no dirigidos**, marca ambos extremos con el mismo valor positivo. Es ideal para analizar flujos de red o circuitos, ya que detalla explícitamente cada conexión individual en una columna propia.

Fragmentos de Código Relevantes

Se usan signos negativos y positivos para indicar la dirección de la flecha:

```
if(esDirigido)
{
    // En el origen va el negativo
    MIncidencia[u][e] = -(peso);

    // En el destino va el positivo
    MIncidencia[v][e] = +(peso);
}
```

Muestra cómo, si no hay dirección, la incidencia es simétrica en valor absoluto:

```
else
{
    // En grafos NO dirigidos:
    // Se marca con el peso en ambos vértices

    MIncidencia[u][e] = peso;
    MIncidencia[v][e] = peso;
}
```

b) 1. Algoritmo de Recorrido: BFS (Breadth-First Search)

Tipo de Grafo: Dirigido/No dirigido, Ponderado/No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** $O(N^2)$
  - La complejidad temporal es cuadrática debido al uso de una Matriz de Adyacencia. Aunque el algoritmo procesa cada vértice una sola vez al sacarlo de la cola, debe iterar obligatoriamente sobre todas las columnas de la matriz (N iteraciones) para identificar a los vecinos adyacentes, resultando en un total de (N)x(N) operaciones.
- **Espacio:** $O(N^2)$
  - La memoria está dominada por la declaración estática de la matriz Independientemente del número de aristas o si el grafo es disperso, se reserva un espacio fijo cuadrático basado en la constante.

Explicación del Funcionamiento

Este algoritmo implementa una búsqueda en anchura utilizando una estructura de datos tipo Cola (FIFO) para gestionar el orden de visita. Comienza insertando el nodo raíz en la cola y marcándolo. En el ciclo principal, extrae el primer elemento y barre toda su fila correspondiente en la matriz de adyacencia. Si encuentra una celda con un valor distinto a NO_EDGE (indicando conexión) y el nodo destino no ha sido procesado previamente, lo marca como visitado y lo añade a la cola. Esto asegura que el grafo se explore nivel por nivel.

Fragmentos de Código Relevantes

Este bloque contiene la implementación completa de la lógica BFS: gestión de la cola, vector de visitados y el ciclo de exploración sobre la matriz de adyacencia.

```
void BFS(int src, int n, int mat[MAXN][MAXN], char elementos[]) {

    vector<bool> visitado(n, false); // vector que guarda si ya visite cada nodo
    queue<int> q;                    // cola que maneja el orden de visita

    visitado[src] = true;            // marco el nodo inicial como visitado
    q.push(src);                     // lo añado a la cola para procesarlo

    cout << "\n=== Recorrido BFS ===\n";

    while (!q.empty()) {             // mientras queden nodos en la cola

        int u = q.front();           // tomo el primer nodo en la cola
        q.pop();                     // lo saco de la cola

        cout << elementos[u] << " "; // imprimo la letra correspondiente al nodo

        for (int v = 0; v < n; v++) { // reviso todos los nodos posibles v

            // si existe una arista u->v y v no fue visitado
            if (mat[u][v] != NO_EDGE && !visitado[v]) {
                visitado[v] = true;  // marco v como visitado
                q.push(v);           // encolo v para procesarlo luego
```

b) 2. Algoritmo de Recorrido: DFS (Depth-First Search)

Tipo de Grafo: Dirigido/No dirigido, Ponderado/No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** $O(N^2)$
  - La complejidad temporal es cuadrática debido a la estructura de datos utilizada (Matriz de Adyacencia). Aunque el algoritmo visita cada vértice una sola vez, dentro de cada llamada recursiva debe iterar obligatoriamente por todas las columnas de la matriz (N iteraciones) para verificar la existencia de aristas hacia otros nodos, resultando en un total de (N)x(N) operaciones.
- **Espacio:** $O(N^2)$
  - El consumo de memoria está dominado por la matriz de adyacencia estática, la cual reserva espacio constante al inicio del programa. Adicionalmente, el uso de la recursividad implica un consumo de memoria en la pila de llamadas del sistema (Stack) proporcional a la profundidad máxima del grafo, que en el peor caso es O(N).

Explicación del Funcionamiento

El algoritmo de Búsqueda en Profundidad explora el grafo siguiendo una estrategia recursiva. Comienza en un nodo raíz, lo marca como visitado y avanza inmediatamente hacia el primer vecino no visitado disponible, profundizando tanto como sea posible en esa rama antes de retroceder (backtracking). A diferencia del BFS que explora por niveles, el DFS utiliza la pila de llamadas del sistema para recordar el camino y retornar al nodo anterior cuando se alcanza un punto ciego, garantizando la visita de todos los nodos conectables.

Fragmento de Código Relevante

Este bloque muestra la implementación recursiva del algoritmo. Se observa la marcación del nodo actual y el ciclo que busca vecinos para realizar la siguiente llamada recursiva, la base del comportamiento "en profundidad".

```
// ------------------------------------------------------------------
// DFS RECURSIVO
// ------------------------------------------------------------------
void DFS(int u, int n, int mat[MAXN][MAXN], vector<bool> &visitado, char elementos[]) {

    visitado[u] = true;        // Marcamos el nodo actual como visitado

    cout << elementos[u] << " ";   // Mostramos la letra del nodo actual

    // Recorremos todos los posibles nodos v
    for(int v = 0; v < n; v++)
        // Si hay una conexion (mat[u][v] != 0) y aun no visitamos al nodo v
        if(mat[u][v] != NO_EDGE && !visitado[v])
            DFS(v, n, mat, visitado, elementos);   // Llamada recursiva
}
```

c) 1. Componentes Conexas (con BFS)

Tipo de Grafo: No dirigido, No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** $O(N^2)$
  - El algoritmo busca visitar cada nodo una sola vez. Sin embargo, al utilizar una matriz de adyacencia, dentro de cada exploración BFS es obligatorio recorrer una fila completa de tamaño N para identificar a los vecinos conectados. Como este proceso se repite en el ciclo principal hasta cubrir todos los vértices, el tiempo total es proporcional al cuadrado del número de nodos.
- **Espacio:** $O(N^2)$
  - El uso de memoria es constante respecto al tamaño máximo definido en el código (MAXN). La matriz de adyacencia grafo[MAXN][MAXN] reserva un espacio fijo en memoria desde el inicio de la ejecución, independientemente de si el grafo tiene pocas o muchas conexiones reales.

Explicación del Funcionamiento

Este algoritmo permite encontrar y separar grupos de nodos que están conectados entre sí pero aislados del resto del grafo. Utiliza un ciclo principal que itera sobre todos los vértices existentes; si encuentra uno que no ha sido visitado previamente, dispara un recorrido BFS completo desde ese punto. Este recorrido marca todos los nodos alcanzables pertenecientes a esa componente conexa. El proceso continúa hasta que todos los nodos del sistema han sido verificados y agrupados.

Fragmento de Código Relevante

Este bloque muestra cómo se envuelve la llamada al BFS dentro de un ciclo iterativo. La condición !visitado[i] es la clave para detectar nuevas componentes que no fueron alcanzadas en recorridos anteriores.

```
// Inicializa todos los nodos como NO visitados
for (int i = 0; i < n; i++)
    visitado[i] = false;

cout << "\n=== Componentes Conexas ===\n";

// Recorre todos los nodos para buscar componentes
for (int i = 0; i < n; i++) {

    // Si el nodo aun no ha sido visitado
    if (!visitado[i]) {

        cout << "CC: ";      // Imprime encabezado de la componente conexa

        bfs(i);              // Llama BFS para explorar esta componente

        cout << "\n";        // Salto de linea para separar componentes
    }
}
```

c) 2. Componentes Conexas (con DFS)

Tipo de Grafo: No dirigido, No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** $O(N^2)$
  - Aunque el algoritmo visita cada nodo una única vez, el uso de una matriz de adyacencia impone un costo fijo en cada paso. Dentro de cada llamada recursiva, es necesario iterar sobre la fila completa (N columnas) para detectar conexiones. Al sumar este proceso para todos los nodos del grafo, el número total de operaciones es proporcional al cuadrado de los vértices.
- **Espacio:** $O(N^2)$
  - La memoria requerida es dominada por la matriz de adyacencia grafo[MAXN][MAXN]. Esta estructura reserva un bloque de memoria contiguo de tamaño fijo basado en la constante MAXN, sin importar si el grafo es denso o disperso. Adicionalmente, la pila de recursión del sistema consume memoria lineal O(N) en el peor caso.

Explicación del Funcionamiento

Este algoritmo identifica grupos de nodos aislados dentro de un grafo utilizando una estrategia de búsqueda en profundidad. El programa itera secuencialmente sobre todos los vértices; cuando encuentra uno que no ha sido marcado como "visitado", inicia una cadena de llamadas recursivas (DFS) que explora profundamente y etiqueta a todos los nodos alcanzables en esa sección específica (Componente Conexa). El ciclo principal continúa buscando nodos no visitados hasta que todo el grafo ha sido segmentado.

Fragmentos de Código Relevantes

A. Exploración Recursiva (DFS) Esta función implementa el recorrido en profundidad. Marca el nodo actual y utiliza un ciclo para revisar la fila de la matriz, lanzando la recursión solo si encuentra una conexión válida hacia un vecino no visitado.

```
// ------------------------------------------------------------
// DFS desde un nodo inicial
// ------------------------------------------------------------
void dfs(int u) {
    visitado[u] = true;      // Marcamos el nodo como visitado
    cout << u << " ";        // Mostramos el nodo visitado

    // Recorremos todos los nodos para ver conexiones desde u
    for (int v = 0; v < n; v++) {
        // Si hay arista de u a v y v no fue visitado, hacemos DFS recursivo
        if (grafo[u][v] == 1 && !visitado[v]) {
            dfs(v);
        }
    }
}
```

B. Identificación de Componentes Este ciclo en la función principal es el encargado de detectar las múltiples componentes. Al verificar la condición !visitado[i], garantiza que se inicie una nueva búsqueda solo para aquellos nodos que quedaron desconectados de exploraciones anteriores.

```
// Recorremos todos los nodos y ejecutamos DFS si no fue visitado
for (int i = 0; i < n; i++) {
    if (!visitado[i]) {
        cout << "CC: "; // Imprimimos etiqueta de Componente Conexa
        dfs(i);          // Ejecutamos DFS desde el nodo i
        cout << "\n";    // Salto de linea al terminar componente
    }
}
```

c) 3. Componentes Conexas (con Union-Find)

Tipo de Grafo: No dirigido, No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** $O(N^2)$
  - Aunque las operaciones internas de Union-Find (findSet y unite) son extremadamente eficientes (casi constante amortizado), el algoritmo principal recorre obligatoriamente toda la matriz de adyacencia mediante dos ciclos anidados para detectar las conexiones existentes. Este barrido completo de la matriz domina la complejidad temporal, resultando en un tiempo proporcional al cuadrado del número de nodos.
- **Espacio:** $O(N^2)$
  - El mayor consumo de memoria proviene de la matriz de adyacencia grafo[MAXN][MAXN], la cual tiene un tamaño fijo cuadrático. Adicionalmente, se utiliza un arreglo lineal padre[MAXN] para gestionar la estructura de conjuntos disjuntos, pero el término cuadrático es el que define la complejidad espacial final.

Explicación del Funcionamiento

Este algoritmo utiliza la técnica de Union-Find para agrupar nodos en conjuntos disjuntos. Inicialmente, cada nodo es su propio "padre" o representante. El programa recorre la matriz de adyacencia y, al encontrar una conexión entre dos nodos, invoca la función unite para fusionar sus conjuntos. La función findSet incluye una optimización llamada "compresión de caminos", la cual actualiza los punteros de los nodos para que apunten directamente a la raíz del conjunto, acelerando futuras búsquedas. Al final, los nodos que comparten la misma raíz pertenecen a la misma componente conexa.

Fragmentos de Código Relevantes

A. Compresión de Caminos (Find) Esta función es clave para la eficiencia de la estructura. Busca el representante del conjunto y, recursivamente, actualiza al padre del nodo actual para que apunte directamente a la raíz, aplanando el árbol de búsqueda.

```
// ------------------------------------------------------------
// Funcion FIND (con compresion de caminos)
// ------------------------------------------------------------
int findSet(int u) {
    if (padre[u] == u) return u;
    return padre[u] = findSet(padre[u]);
}
```

B. Unión de Conjuntos basada en la Matriz Este bloque muestra cómo el algoritmo procesa la entrada. Itera sobre cada celda de la matriz y, si detecta una arista (valor 1), une inmediatamente los subconjuntos de los nodos involucrados, construyendo dinámicamente las componentes.

```
// Unir nodos conectados
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (grafo[i][j] == 1) {
            unite(i, j);
        }
    }
}
```

c) 4. Componentes Fuertemente Conexas (Algoritmo de Gabow)

Tipo de Grafo: Dirigido, No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** $O(N^2)$
  - El algoritmo de Gabow es eficiente y teóricamente puede ejecutarse en tiempo lineal relativo a las aristas. Sin embargo, debido a la implementación con matriz de adyacencia, el paso de buscar vecinos requiere recorrer una fila completa de tamaño N para cada nodo visitado. Esto eleva el costo total de operación al cuadrado del número de nodos.
- **Espacio:** $O(N^2)$
  - El consumo de memoria es cuadrático debido a la reserva estática de la matriz `matriz[MAXN][MAXN]`. Además, se utiliza espacio lineal O(N) para las dos pilas (`S` y `P`) y el arreglo de tiempos `pre[]`, pero el término dominante sigue siendo la matriz.

Explicación del Funcionamiento

Este algoritmo identifica Componentes Fuertemente Conexas (SCC) en un grafo dirigido realizando una sola búsqueda en profundidad (DFS). Utiliza dos pilas: la pila `S`, que almacena todos los nodos del camino actual, y la pila `P`, que ayuda a determinar la raíz de cada componente conexa. Cuando el algoritmo detecta un enlace hacia un nodo ancestro ya visitado, "colapsa" la pila `P` para fusionar el ciclo. Al finalizar la recursión de un nodo, si este permanece en el tope de `P`, se confirma que es la raíz de una SCC y se extraen los nodos correspondientes de `S`.

Fragmentos de Código Relevantes

A. Gestión de Pilas y "Rebobinado" Este bloque muestra la esencia del algoritmo de Gabow: insertar el nodo en ambas pilas al descubrirlo y gestionar el caso de retroceso (back-edge) cuando se encuentra un vecino ya visitado con un tiempo de descubrimiento menor.

```
// Asignar tiempo de descubrimiento al nodo
pre[u] = preCont++;

// Meter nodo en la pila principal
S.push(u);

// Tambien meter el nodo en la pila auxiliar
P.push(u);

// Recorrer todos los posibles destinos desde u
for (int v = 0; v < n; v++) {

    // Si hay arista de u hacia v
    if (matriz[u][v] == 1) {

        // Si v aun no ha sido visitado
        if (pre[v] == -1) {

            // Se sigue explorando recursivamente
            gabowDFS(v);
        }
        // Si v ya fue descubierto y su tiempo es menor al de u
        else if (pre[v] < pre[u]) {

            // Se hace "rebobinado" de pila P para mantener limites correctos
            while (!P.empty() && pre[P.top()] > pre[v])
                P.pop();
        }
    }
}
```

B. Extracción de la Componente Conexa Este fragmento se ejecuta al terminar de procesar los hijos de un nodo. Si el nodo actual `u` está en el tope de la pila auxiliar `P`, significa que se ha completado una componente fuertemente conexa, por lo que se procede a imprimirla vaciando la pila `S` hasta encontrar a `u`.

```
// Si el tope de la pila P coincide con u, encontramos una SCC
if (!P.empty() && P.top() == u) {

    cout << "\nComponente fuertemente conexa: ";

    int v;

    // Extraemos nodos de la pila S hasta volver a u
    do {
        v = S.top();  // Obtener nodo del tope
        S.pop();      // Quitarlo de la pila
        cout << elementos[v] << " "; // Imprimir su letra

    } while (v != u);   // Se detiene cuando regresamos al nodo inicial u

    cout << "\n";

    P.pop();   // Quitar tambien el nodo de la pila auxiliar
}
```

c) 5. Componentes Fuertemente Conexas (Algoritmo de Kosaraju)

Tipo de Grafo: Dirigido, No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** $O(N^2)$
  - Si bien el algoritmo de Kosaraju es teóricamente lineal O(V+E) usando listas de adyacencia, esta implementación utiliza **Matrices de Adyacencia**. Esto obliga a que tanto el primer DFS, la transposición del grafo y el segundo DFS deban iterar por todas las columnas de la matriz para cada nodo, resultando en tres operaciones de orden N^2 lo que simplifica la complejidad total a cuadrática.
- **Espacio:** $O(N^2)$
  - El uso de memoria es alto debido a la necesidad de almacenar dos matrices completas de tamaño MAXN x MAXN: una para el grafo original (grafo) y otra para el grafo invertido (transpuesto).

Explicación del Funcionamiento

Este algoritmo encuentra las Componentes Fuertemente Conexas (SCC) mediante dos pasadas de profundidad (DFS). En la primera pasada, recorre el grafo original y almacena los nodos en una pila según su orden de finalización (post-orden). Posteriormente, genera un "grafo transpuesto" invirtiendo la dirección de todas las aristas. Finalmente, procesa los nodos extrayéndolos de la pila y ejecuta un segundo DFS sobre el grafo transpuesto; cada conjunto de nodos alcanzable en esta segunda etapa constituye una SCC aislada.

Fragmentos de Código Relevantes

A. Primera Pasada (Ordenamiento en Pila)

Realiza un DFS para apilar los nodos. La línea clave es pila.push(u), que asegura que los nodos se procesen después en orden inverso a su finalización.

```
// ====================================================================
// PRIMER DFS (dfs1): sirve para llenar la pila en orden de salida
// ====================================================================
void dfs1(int u) {
    visitado[u] = true; // Marcar nodo como visitado

    // Explorar todos los nodos v alcanzables desde u
    for (int v = 0; v < n; v++)
        if (grafo[u][v] == 1 && !visitado[v]) // Si hay arista y no visitado
            dfs1(v);                          // Llamada recursiva

    pila.push(u);  // Agregar nodo a la pila al terminar sus recorridos
}
```

B. Transposición del Grafo

Invierte la dirección de las aristas. Lo que antes era una conexión de j a i, ahora se guarda como i a j en la matriz auxiliar.

```
// ========================================================================
// TRANSPONER GRAFO: invierte todas las direcciones de las aristas
// ========================================================================
void transponerGrafo() {
    for (int i = 0; i < n; i++)            // Recorrer filas
        for (int j = 0; j < n; j++)        // Recorrer columnas
            transpuesto[i][j] = grafo[j][i]; // Intercambiar j->i
}
```

C. Segunda Pasada (Identificación de SCC)

Utiliza la pila llena en el paso A para guiar el orden de exploración sobre el grafo transpuesto. Cada vez que se logra entrar al if (!visitado[u]), se ha encontrado una nueva componente conexa.

```
// Vaciar pila para determinar SCCs
while (!pila.empty()) {
    int u = pila.top(); // Obtener cima de la pila
    pila.pop();         // Quitarla

    // Si el nodo no ha sido visitado en la segunda pasada,
    // entonces inicia una nueva SCC
    if (!visitado[u]) {
        cout << "SCC: ";
        dfs2(u);        // Explorar e imprimir los nodos de la SCC
        cout << "\n";
    }
}
```

c) 6. Componentes Fuertemente Conexas (Algoritmo de Tarjan)

Tipo de Grafo: Dirigido, No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** $O(N^2)$
  - El algoritmo de Tarjan está diseñado para ser lineal O(V+E) si se usan listas de adyacencia. Sin embargo, esta implementación se basa en una **Matriz de Adyacencia**, lo que obliga al algoritmo a recorrer toda la fila de posibles vecinos (tamaño N) por cada nodo visitado en el DFS. Esto convierte la eficiencia lineal en una complejidad cuadrática.
- **Espacio:** $O(N^2)$
  - El consumo de memoria es constante respecto a la capacidad máxima del sistema (MAXN), dominado por la matriz matriz[MAXN][MAXN]. Además, se utilizan arreglos auxiliares lineales para ids, low y la pila, pero el término cuadrático de la matriz prevalece en el análisis asintótico.

Explicación del Funcionamiento

Este algoritmo localiza Componentes Fuertemente Conexas (SCC) en una sola pasada de profundidad (DFS). Mantiene dos valores por nodo: ids (tiempo de descubrimiento) y low (el identificador más bajo alcanzable desde ese nodo). A medida que explora, apila los nodos visitados. Cuando la recursión retorna, actualiza el valor low basándose en los hijos. Si al terminar de procesar un nodo su low es igual a su ids, significa que ese nodo es la "raíz" de una componente conexa, por lo que procede a desapilar todos los elementos acumulados hasta ese punto.

Fragmentos de Código Relevantes

A. Cálculo de Low-Link y Recursión Este bloque muestra cómo se asignan los identificadores iniciales y cómo se actualiza el valor low al volver de la recursión (si el vecino no fue visitado) o al encontrar una arista hacia atrás (si el vecino ya estaba en la pila).

```
ids[u] = low[u] = idActual++;     // Asigna id inicial y low iguales
pilaTarjan.push(u);               // Coloca u en la pila
enPila[u] = true;                 // Marca que u esta en la pila

// ----------------------
// Explorar vecinos de u
// ----------------------
for(int v = 0; v < n; v++) {

    if(matriz[u][v] == 1) {       // Si hay arista u -> v

        if(ids[v] == -1) {        // Si v aun no fue visitado
            tarjanDFS(v);         // Llama recursivamente a Tarjan en v
            low[u] = min(low[u], low[v]);    // Actualiza low de u
        }
        else if(enPila[v]) {      // Si v esta en la pila, es parte del SCC actual
            low[u] = min(low[u], ids[v]);    // Actualiza low usando id de v
```

B. Identificación y Extracción de SCC Cuando se confirma que el nodo actual u es la raíz de una componente (su enlace más bajo apunta a sí mismo), se extraen todos los nodos de la pila hasta encontrar nuevamente a u, formando así el grupo fuertemente conexo.

```
// ------------------------------------------
// Si u es una raiz de SCC (low[u] == ids[u])
// ------------------------------------------
if(ids[u] == low[u]) {

    cout << "\nComponente #" << ++sccCount << ": ";

    // Extraer nodos del SCC desde la pila
    while(true) {

        int v = pilaTarjan.top();   // Obtiene el nodo superior
        pilaTarjan.pop();           // Lo quita de la pila
        enPila[v] = false;          // Marca que ya no esta en la pila

        cout << elementos[v] << " "; // Imprime el nodo encontrado

        if(v == u)                  // Si llegamos al nodo raiz, detener
            break;
    }

    cout << "\n";                   // Salto de linea
}
```

d) 1. Camino Más Corto: Backtracking con Poda

Tipo de Grafo: Dirigido/No Dirigido, Ponderado/No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(N!)
  - La complejidad temporal en el peor de los casos es factorial. Al tratarse de una búsqueda exhaustiva que intenta explorar todos los caminos posibles sin repetir nodos (camino simple), si el grafo es denso (como un grafo completo), el número de rutas posibles crece factorialmente respecto al número de nodos (N). La "poda" ayuda a reducir significativamente el tiempo promedio al descartar ramas inútiles, pero no cambia el límite superior del peor caso.
- **Espacio:** O(N^2)
  - El consumo de memoria está dominado por la **Matriz de Adyacencia** mat[MAXN][MAXN], que ocupa un espacio fijo cuadrático. Adicionalmente, la pila de recursión y los vectores de camino consumen memoria lineal O(N), pero esto es despreciable comparado con la matriz.

Explicación del Funcionamiento

Este algoritmo utiliza una estrategia recursiva para explorar rutas desde un nodo origen hasta un destino. A diferencia de un recorrido simple, mantiene registro del costo acumulado y del "mejor costo" encontrado hasta el momento. La característica clave es la **Poda**: antes de avanzar a un nuevo nodo, verifica si el costo actual ya supera al mejor costo registrado. Si es así, detiene inmediatamente esa rama de exploración (backtracking), ahorrando tiempo de cómputo. Utiliza un arreglo de visitados para evitar ciclos y garantizar que los caminos sean simples.

Fragmentos de Código Relevantes

A. Condición de Poda (Optimization)

Esta es la línea que diferencia un backtracking puro de uno con "Branch and Bound". Si el camino actual ya es más caro que uno completo encontrado anteriormente, se descarta inmediatamente.

```
// Funcion recursiva de backtracking con poda
void backtrack(int u, int dest, int currCost, vector<int> &path, vector<bool> &visited){
    if(currCost>=bestCost) return; // poda: no explorar caminos mas caros
    if(u==dest){ // llegamos al destino
        bestCost=currCost;
        bestPath=path;
        return;
    }
```

B. Exploración y Vuelta Atrás (Backtracking)

Este bloque muestra el núcleo del algoritmo: marcar el nodo actual, realizar la llamada recursiva y, crucialmente, "desmarcarlo" (visited[v]=false y path.pop_back()) al regresar. Esto permite que el nodo pueda ser reutilizado en otros caminos alternativos que se exploren posteriormente.

```
for(int v=0;v<n;v++){
    if(mat[u][v]!=NO_EDGE && !visited[v]){ // si hay arista y no visitado
        int w=weighted? mat[u][v]:1;       // costo de la arista
        visited[v]=true;                   // marcamos como visitado
        path.push_back(v);                 // agregamos al camino
        backtrack(v,dest,currCost+w,path,visited); // llamada recursiva
        path.pop_back();                   // desmarcamos el nodo
        visited[v]=false;
```

d) 2. Caminos Más Cortos: Algoritmo de Bellman-Ford

Tipo de Grafo: Dirigido/No Dirigido, Ponderado (acepta pesos negativos)/No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(N^3)
  - El algoritmo estándar de Bellman-Ford opera teóricamente en O(V * E). Sin embargo, esta implementación utiliza una Matriz de Adyacencia. Para procesar todas las aristas en cada iteración, el algoritmo debe recorrer la matriz completa (N * N). Dado que este proceso se repite N-1 veces (el número máximo de aristas en un camino simple), se generan tres ciclos anidados, resultando en una complejidad cúbica.
- **Espacio:** O(N^2)
  - La estructura principal es la matriz de adyacencia mat[MAXN][MAXN], la cual reserva memoria constante al inicio de la ejecución. Se utiliza un vector adicional dist de tamaño N, pero el término cuadrático de la matriz domina el análisis de espacio.

Explicación del Funcionamiento

Este algoritmo calcula la ruta más corta desde un nodo origen hacia todos los demás, siendo capaz de manejar aristas con pesos negativos. Funciona mediante la técnica de "relajación": itera N-1 veces sobre todo el grafo, intentando reducir la distancia estimada hacia cada destino si encuentra un camino más barato a través de una arista intermedia. Tras completar estas iteraciones, realiza una pasada final; si todavía es posible reducir alguna distancia, determina que el grafo contiene un "ciclo negativo", lo que hace imposible definir un camino mínimo estable.

Fragmentos de Código Relevantes

A. Relajación de Aristas Este bloque muestra los tres ciclos anidados. El ciclo externo `k` asegura la propagación de las distancias, mientras que los ciclos internos `u` y `v` recorren la matriz para intentar mejorar (relajar) los caminos existentes.

```
// ----------------------------
// RELAJACIÓN DE ARISTAS
// Repetimos n-1 veces: propiedad de Bellman-Ford
// ----------------------------
for (int k = 0; k < n-1; ++k) {
    for (int u = 0; u < n; ++u) {
        for (int v = 0; v < n; ++v) {
            if (mat[u][v] != NO_EDGE) {   // si hay arista u->v
                int w = weighted ? mat[u][v] : 1;
                if (dist[u] != INF && dist[u] + w < dist[v])
                    dist[v] = dist[u] + w; // relajamos la arista
            }
        }
    }
}
```

B. Detección de Ciclos Negativos Este segmento valida la integridad de los resultados. Si después de las N-1 iteraciones todavía se cumple la condición `dist[u] + w < dist[v]`, se concluye la existencia de un ciclo negativo.

```
// ----------------------------
// DETECCIÓN DE CICLOS NEGATIVOS
// Si aún se puede relajar alguna arista, hay ciclo negativo accesible
// ----------------------------
bool cicloNegativo = false;
for (int u = 0; u < n; ++u) {
    for (int v = 0; v < n; ++v) {
        if (mat[u][v] != NO_EDGE) {
            int w = weighted ? mat[u][v] : 1;
            if (dist[u] != INF && dist[u] + w < dist[v]) {
                cicloNegativo = true; // ciclo negativo detectado
                break;
            }
        }
    }
    if (cicloNegativo) break;
}
```

d) 3. Caminos Más Cortos: Algoritmo de Floyd-Warshall

Tipo de Grafo: Dirigido/No Dirigido, Ponderado/No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(N^3)
  - La complejidad está determinada por la estructura de tres ciclos for anidados que son característicos de este algoritmo. El ciclo externo itera sobre cada posible nodo intermedio (k), mientras que los dos ciclos internos iteran sobre cada par posible de nodos origen (i) y destino (j). Dado que cada ciclo recorre los N nodos del grafo, el número total de operaciones es $N \times N \times N$.
- **Espacio:** O(N^2)
  - Se utilizan dos matrices bidimensionales principales de tamaño MAXN x MAXN: mat para la entrada de adyacencia y dist para almacenar las distancias mínimas calculadas. Esto resulta en un consumo de memoria cuadrático fijo, independientemente de la cantidad de aristas (densidad) del grafo.

Explicación del Funcionamiento

Este es un algoritmo de programación dinámica diseñado para encontrar el camino más corto entre **todos los pares** de nodos en un grafo. Comienza inicializando una matriz de distancias donde se registran los pesos de las aristas directas conocidas y se marca como infinito aquello que no está conectado directamente. Posteriormente, el algoritmo intenta mejorar la ruta entre cada par de nodos (i, j) verificando si pasar por un nodo intermedio k resulta en un costo menor que el camino conocido actualmente. Si la ruta i -> k -> j es más corta, se actualiza la matriz de distancias.

Fragmentos de Código Relevantes

A. Inicialización de la Matriz de Distancias

Antes de ejecutar la lógica principal, se debe preparar la matriz dist. Se establecen las distancias a sí mismo en 0, se copian los pesos de las aristas existentes y se marcan con INF los pares sin conexión directa.

```
int dist[MAXN][MAXN];
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++){
        if(i==j) dist[i][j]=0; // distancia a si mismo = 0
        else if(mat[i][j]!=NO_EDGE) dist[i][j]=weighted?mat[i][j]:1;
        else dist[i][j]=INF; // no hay arista directa
    }
```

B. Núcleo del Algoritmo (Tres Ciclos Anidados)

Este bloque representa la lógica de programación dinámica. El nodo k actúa como pivote; si el camino desde i hasta k más el camino desde k hasta j es menor que la distancia actual entre i y j, se actualiza el valor.

```
// algoritmo Floyd-Warshall: actualizar distancias mínimas
for(int k=0;k<n;k++)
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            if(dist[i][k]!=INF && dist[k][j]!=INF && dist[i][k]+dist[k][j]<dist[i][j])
                dist[i][j]=dist[i][k]+dist[k][j];
```

e) 1. Verificación de Árbol (DFS)

Tipo de Grafo: Dirigido/No Dirigido, Ponderado/No Ponderado (se ignora peso).

Análisis de Complejidad (Big-O)

- **Tiempo:** O(V + E)
  - El algoritmo realiza dos verificaciones principales mediante DFS: detección de ciclos y validación de conectividad. Al utilizar **Listas de Adyacencia** (vector<int> lista[]), cada nodo y cada arista se visitan un número constante de veces. Por tanto, el tiempo total es proporcional a la suma de vértices y aristas.
- **Espacio:** O(V + E)
  - La memoria principal se utiliza para almacenar el grafo en las listas de adyacencia (V + E). Adicionalmente, se requieren arreglos auxiliares lineales (O(V)) para el control de visitas (visitado[]) y la pila de recursión (enRecursion[]).

Explicación del Funcionamiento

Este algoritmo determina si un grafo cumple con las propiedades matemáticas de un árbol: debe ser **conexo** (todos los nodos están unidos) y **acíclico** (no contiene ciclos cerrados).

1. **Detección de Ciclos:** Utiliza una búsqueda en profundidad (DFS).
   - En grafos **no dirigidos**, verifica si se encuentra una arista hacia un nodo ya visitado que no sea el padre inmediato.
   - En grafos **dirigidos**, utiliza una "pila de recursión" para detectar si se regresa a un nodo que está siendo procesado actualmente en la rama de exploración.

2. **Conectividad:** Realiza un segundo DFS desde un nodo raíz arbitrario (nodo 0) y verifica si logró visitar todos los N nodos. Si algún nodo queda sin visitar, el grafo es disconexo y, por tanto, no es un árbol.

Fragmentos de Código Relevantes

A. Detección de Ciclos (Caso Dirigido)

Este bloque es crucial. Utiliza el arreglo enRecursion para rastrear los nodos activos en la ruta actual. Si encuentra un vecino que ya está en recursión, confirma la existencia de un ciclo.

```
bool dfsDirigido(int u)
{
    visitado[u] = true;        // Marcamos el nodo como visitado
    enRecursion[u] = true;     // Lo agregamos a la pila de recursión

    int i;
    for(i = 0; i < lista[u].size(); i++)
    {
        int v = lista[u][i];  // Vecino v de u

        if(!visitado[v])      // Si v NO ha sido visitado
        {
            if(dfsDirigido(v))   // Llamada recursiva
                return true;     // Se detectó ciclo
        }
        else if(enRecursion[v])  // Si ya estaba en la pila ? ciclo
        {
            return true;
        }
    }

    enRecursion[u] = false;    // Sacamos a u de la pila
    return false;              // No hubo ciclo
}
```

B. Verificación Final (Función Maestra)

Esta función orquesta las dos validaciones. Si falla la prueba de ciclos o la prueba de conectividad, retorna false inmediatamente.

```
if(esDirigido)
{
    // 1. Detectar ciclos
    for(i = 0; i < N; i++)
    {
        visitado[i] = false;
        enRecursion[i] = false;
    }

    for(i = 0; i < N; i++)
    {
        if(!visitado[i])
            if(dfsDirigido(i))
                return false;  // Tiene ciclo ? NO árbol
    }

    // 2. Verificar conectividad
    for(i = 0; i < N; i++)
        visitado[i] = false;

    dfsConectividad(0);   // Empezamos desde nodo 0

    for(i = 0; i < N; i++)
        if(!visitado[i])
            return false; // No es conexo ? NO árbol

    return true;          // Es árbol dirigido
}
```

e) 2. Verificación de Árbol (BFS y Grados)

Tipo de Grafo: Dirigido/No Dirigido, Ponderado/No Ponderado (se ignoran pesos).

Análisis de Complejidad (Big-O)

- **Tiempo:** O(V + E)
  - Esta implementación utiliza **Listas de Adyacencia** (vector<int> lista[]). El algoritmo BFS visita cada vértice y cada arista exactamente una vez. Las verificaciones adicionales (conteo de grados o validación de E = N-1) son operaciones lineales que recorren los nodos (V) o se realizan en tiempo constante. Por tanto, la complejidad es lineal respecto al tamaño del grafo.
- **Espacio:** O(V + E)
  - El uso de memoria es óptimo para grafos dispersos gracias a las listas de adyacencia, que solo almacenan las conexiones existentes. También se utilizan arreglos auxiliares lineales (O(V)) para visitado[], indegreeArr[] y la cola del BFS.

Explicación del Funcionamiento

Este algoritmo verifica si un grafo cumple las propiedades estructurales de un árbol utilizando Búsqueda en Anchura (BFS) y conteo de aristas/grados.

1. **Caso No Dirigido:** Verifica primero la propiedad matemática fundamental de que un árbol de N nodos debe tener exactamente N-1 aristas. Luego, ejecuta un BFS para asegurar que el grafo sea conexo (todos los nodos accesibles).
2. **Caso Dirigido:** Verifica la topología contando los "grados de entrada" (indegree). Un árbol dirigido debe tener una única raíz (indegree 0) y todos los demás nodos deben tener exactamente un padre (indegree 1). Finalmente, valida la conectividad desde la raíz.

Fragmentos de Código Relevantes

A. Verificación No Dirigida (Aristas y Conectividad)

Este bloque valida la condición necesaria (E = N-1) y luego lanza el BFS para confirmar que no hay nodos aislados (conectividad).

```
bool esArbolNoDirigido()
{
    if(E != N - 1)
        return false;

    for(int i = 0; i < N; i++)
        visitado[i] = false;

    bfs(0);

    for(int i = 0; i < N; i++)
        if(!visitado[i])
            return false;

    return true;
}
```

B. Verificación Dirigida (Indegree y Raíz)

Este segmento es clave para árboles dirigidos. Cuenta cuántas raíces existen (debe ser 1) y verifica que el resto de los nodos tengan solo una arista entrante, garantizando la ausencia de ciclos dirigidos y la estructura jerárquica.

```
for(int i = 0; i < N; i++)
{
    if(indegreeArr[i] == 0)
    {
        raiz = i;
        conteoRaices++;
    }
}

if(conteoRaices != 1)
    return false;

int correctos = 0;

for(int i = 0; i < N; i++)
    if(i != raiz && indegreeArr[i] == 1)
        correctos++;
```

e) 3. Verificación de Árbol (DFS, Grados y N-1)

Tipo de Grafo: Dirigido/No Dirigido, Ponderado/No Ponderado (se ignoran pesos).

Análisis de Complejidad (Big-O)

- **Tiempo:** O(V + E)
  - El algoritmo emplea búsquedas en profundidad (DFS) que visitan cada nodo y cada arista una vez para detectar ciclos y verificar conectividad. En el caso de árboles dirigidos, se añade un paso lineal (O(V)) para contar grados de entrada y verificar la existencia de una raíz única. Al usar listas de adyacencia, todas las operaciones se mantienen lineales.
- **Espacio:** O(V + E)
  - La memoria principal se destina a las listas de adyacencia. Adicionalmente, se utilizan arreglos auxiliares de tamaño proporcional a los vértices (O(V)) para el control de visitas (visitado), detección de ciclos en curso (enProceso) y conteo de grados (indegree).

Explicación del Funcionamiento

Este enfoque combina tres criterios matemáticos para validar si un grafo es un árbol:

1. **Conectividad (Vía DFS):** Verifica que todos los nodos sean accesibles desde un punto de partida (la raíz en grafos dirigidos o un nodo arbitrario en no dirigidos).
2. **Ciclicidad (Vía DFS):** Detecta ciclos cerrados. En grafos dirigidos usa una pila de recursión (enProceso); en no dirigidos, verifica que no se regrese al nodo padre.
3. **Propiedades Numéricas:**
   - **No Dirigido:** Debe cumplirse estrictamente que el número de aristas válidas del árbol DFS sea igual a N-1.
   - **Dirigido:** Debe existir exactamente una raíz (indegree 0) y todos los demás nodos deben tener exactamente un padre (indegree 1). También se verifica E = N-1.

Fragmentos de Código Relevantes

A. Validación Numérica del Árbol No Dirigido

Este bloque es la validación final. Si el conteo de aristas descubiertas por el DFS no coincide con N-1, o si se detectó un ciclo explícito (marcado con 999999), se descarta la estructura de árbol.

```
// 2. Si detectamos ciclo
if(aristasDFS == 999999)
    return false;

// 3. Si las aristas del DFS no son N-1 = no es árbol
if(aristasDFS != N - 1)
    return false;
```

B. Validación de Grados (Árbol Dirigido)

Este segmento asegura la jerarquía correcta. Busca una única raíz y garantiza que ningún nodo tenga múltiples padres, lo cual rompería la definición de árbol.

```
for(i = 0; i < N; i++)
{
    if(indegree[i] == 0)
    {
        raiz = i;
        cantidadRaices++;
    }
    else if(indegree[i] > 1)
    {
        return false; // Nodo con más de un padre = NO árbol
    }
}
```

f) 1. Árbol de expansión mínima: Kruskal

Tipo de Grafo: No Dirigido, Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(E \log E)
  - La complejidad está dominada por el paso de ordenamiento de las aristas. El algoritmo necesita ordenar todas las conexiones disponibles por su peso de menor a mayor. Utilizando un algoritmo de ordenamiento eficiente (como el sort de la librería estándar), esto toma tiempo logarítmico respecto al número de aristas. Las operaciones de conjunto (Union-Find) son casi constantes y no afectan el orden asintótico final.
- **Espacio:** O(V + E)
  - Se requiere memoria lineal para almacenar la lista completa de aristas (E) y los arreglos auxiliares necesarios para la estructura de conjuntos disjuntos (padre y rangoUF), cuyo tamaño depende del número de vértices (V).

Explicación del Funcionamiento

Kruskal es un algoritmo de tipo voraz (greedy) diseñado para encontrar el Árbol de Expansión Mínima (MST) de un grafo. Su lógica consiste en procesar las aristas en orden ascendente según su peso. Para cada arista, verifica si los dos nodos que conecta ya pertenecen al mismo componente conexo utilizando una estructura de datos Union-Find. Si los nodos están desconectados, se añade la arista al MST y se unen sus conjuntos; si ya estaban conectados, la arista se descarta para evitar la formación de ciclos.

Fragmentos de Código Relevantes

A. Criterio de Ordenamiento

Esta función es fundamental para la estrategia del algoritmo, ya que garantiza que siempre se intenten procesar primero las conexiones más económicas.

```
// comparador usado por sort para ordenar aristas por peso ascendente
bool compararAristas(const Arista &A, const Arista &B) {
    // devolver true si el peso de A es menor que el peso de B
    return A.w < B.w;
}
```

B. Selección y Unión de Aristas

Este bloque contiene el núcleo del algoritmo: recorre las aristas ordenadas, verifica si forman un ciclo usando encontrarPadre y, si es válida, une los componentes con unir.

```
// recorrer todas las aristas ordenadas
for(int i = 0; i < E; i++) {
    // obtener el extremo u de la arista i
    int u = aristas[i].u;
    // obtener el extremo v de la arista i
    int v = aristas[i].v;

    // si u y v pertenecen a conjuntos distintos no forman ciclo
    if(encontrarPadre(u) != encontrarPadre(v)) {
        // agregar la arista i al arreglo mst en la posicion contador
        mst[contador] = aristas[i];
        // incrementar el contador de aristas del MST
        contador++;
        // sumar el peso de esta arista al peso total
        pesoTotal += aristas[i].w;
        // unir los conjuntos de u y v en union find
        unir(u, v);
    }
}
```

f) 2. Árbol de expansión mínima: Prim

Tipo de Grafo: No Dirigido, Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(V^2)
  - La complejidad está determinada por la estructura de doble ciclo que busca el nodo más cercano no visitado. El ciclo externo se ejecuta V veces (una para cada nodo agregado al MST). Dentro de este, hay un ciclo interno que recorre todos los nodos para encontrar el mínimo valor en el arreglo dist (O(V)). Esto resulta en (V)x(V) operaciones.
- **Espacio:** O(V^2)
  - Se utilizan matrices de adyacencia (adyNodo y adyPeso) de tamaño fijo 100X100 para almacenar el grafo, lo que ocupa un espacio constante cuadrático independientemente de la densidad del grafo real. Además, se usan arreglos lineales auxiliares dist, visitado y padre.

Explicación del Funcionamiento

El algoritmo de Prim construye el Árbol de Expansión Mínima (MST) de manera incremental comenzando desde un nodo raíz arbitrario (nodo 0). Mantiene un conjunto de vértices ya incluidos en el MST y un conjunto de candidatos. En cada paso, selecciona de forma voraz (greedy) el nodo fuera del MST que tenga la arista de conexión más barata con cualquiera de los nodos ya incluidos. Una vez seleccionado, se marca como visitado y se actualizan las distancias de sus vecinos si se encuentra un camino más corto hacia ellos.

Fragmentos de Código Relevantes

A. Selección Voraz del Nodo Más Cercano

Este bloque es el corazón de la complejidad cuadrática. Busca linealmente en el arreglo dist cuál es el siguiente nodo óptimo para añadir al árbol.

```
for(i = 0; i < N; i++)
{
    if(!visitado[i] && dist[i] < mejor)
    {
        mejor = dist[i];
        u = i;
    }
}
```

B. Actualización de Distancias (Relajación)

Una vez añadido el nodo u, se revisan sus vecinos. Si conectarse a un vecino v a través de u es más barato que la mejor opción conocida hasta el momento, se actualiza su distancia y su padre.

```
for(int k = 0; k < grado[u]; k++)
{
    int v = adyNodo[u][k];
    int w = adyPeso[u][k];

    if(!visitado[v] && w < dist[v])
    {
        dist[v] = w;
        padre[v] = u;
    }
}
```

f) 3. Árbol de expansión máxima: Reverse-Kruskal

Tipo de Grafo: No Dirigido, Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(ElogE+E·(V+E))
  - El ordenamiento inicial toma ElogE. El núcleo del algoritmo es el ciclo que recorre las E aristas; en cada iteración, elimina una arista y verifica la conectividad ejecutando un DFS completo, lo que cuesta O(V+E). En el peor de los casos, la complejidad total es dominada por E×E, resultando en O(E2).
- **Espacio:** O(V2)
  - Se utiliza una matriz de adyacencia de tamaño 100×100 para representar el grafo temporalmente durante las verificaciones de conectividad, lo que implica un uso de memoria cuadrático constante independientemente del tamaño real del grafo. Además, se almacenan las aristas en un arreglo lineal.

Explicación del Funcionamiento

El algoritmo "Reverse-Kruskal" (o Reverse-Delete) construye el Árbol de Expansión Mínima (o Máxima, dependiendo del ordenamiento) comenzando con el grafo completo y eliminando aristas redundantes.

1. **Ordenamiento:** Ordena todas las aristas de mayor a menor peso (para MST).
2. **Eliminación Tentativa:** Recorre las aristas en orden descendente. Para cada arista, la elimina temporalmente del grafo.
3. **Verificación de Conectividad:** Ejecuta un DFS para comprobar si el grafo sigue siendo conexo (todos los nodos accesibles) sin esa arista.
    - Si el grafo se desconecta, la arista es un "puente" necesario y se restaura.
    - Si el grafo permanece conexo, la arista es redundante (forma parte de un ciclo) y se elimina permanentemente.

Fragmentos de Código Relevantes

A. Ordenamiento Descendente A diferencia de Kruskal estándar, aquí ordenamos de mayor a menor para intentar eliminar primero las aristas más costosas.

```
bool ordenarDesc(const Arista &A, const Arista &B)
{
    return A.w > B.w;
}
```

B. Ciclo de Eliminación y Verificación Este bloque muestra la lógica de "borrar, verificar conectividad y restaurar si es necesario".

```
for(int i = 0; i < E; i++)
{
    int a = aristas[i].u;
    int b = aristas[i].v;

    matriz[a][b] = false;
    matriz[b][a] = false;

    for(int k = 0; k < N; k++)
        visitado[k] = false;

    dfs(0, matriz);

    bool conectado = true;
```

```
    for(int k = 0; k < N; k++)
    {
        if(!visitado[k])
        {
            conectado = false;
            break;
        }
    }

    if(!conectado)
    {
        matriz[a][b] = true;
        matriz[b][a] = true;
    }
    else
    {
        eliminar[i] = true;
    }
}
```

g) 1. Grafo Bipartito

Tipo de Grafo: No Dirigido, No Ponderado (se ignoran pesos).

Análisis de Complejidad (Big-O)

- **Tiempo:** O(V + E)
  - El algoritmo utiliza una Búsqueda en Anchura (BFS) para recorrer el grafo. En el peor de los casos, cada vértice (V) entra en la cola una sola vez y cada arista (E) es revisada una vez (o dos, dependiendo de la implementación de la lista) para verificar el color del vecino. Por tanto, el tiempo es lineal respecto al tamaño del grafo.
- **Espacio:** O(V + E)
  - La estructura principal es una lista de adyacencia implementada con vector<vector<int>>, que ocupa espacio proporcional a los vértices y aristas. Adicionalmente, se utilizan estructuras auxiliares lineales (O(V)) como el vector de colores y la cola para el BFS.

Explicación del Funcionamiento

Este algoritmo determina si un grafo es bipartito intentando colorearlo con dos colores (0 y 1) de tal manera que ninguna arista conecte dos nodos del mismo color.

Utiliza un recorrido BFS: comienza asignando un color a un nodo inicial y el color opuesto a sus vecinos. Si durante el proceso encuentra un vecino que ya ha sido visitado y tiene el mismo color que el nodo actual, se detecta un conflicto (ciclo de longitud impar), concluyendo que el grafo no es bipartito. El algoritmo itera sobre todos los nodos para asegurar que funcione incluso en grafos disconexos.

Fragmentos de Código Relevantes

A. Lógica de Coloreo y Detección de Conflictos

Este es el núcleo del algoritmo. Si el vecino no tiene color, se le asigna el opuesto (1 - color[u]). Si ya tiene color y es igual al actual, se retorna false inmediatamente.

```
// Revisamos a todos los "vecinos" (nodos conectados) del nodo actual 'u'.
// Usamos size_t solo porque es el tipo correcto para contar elementos, pero es como un contador
for (size_t i = 0; i < grafo[u].size(); ++i) {
    int vecino = grafo[u][i];

    // CONDICION 1: VECINO SIN COLOREAR
    if (color[vecino] == -1) {
        // Le asignamos el color opuesto al nodo actual.
        // Esto garantiza la alternancia de colores (A, B, A, B...).
        color[vecino] = 1 - color[u];

        // Anadimos el vecino a la cola para seguir el recorrido.
        cola.push(vecino);
    }
    // CONDICION 2: ¡CONFLICTO DE COLORES!
    // Si el vecino ya esta coloreado y su color es IGUAL al del nodo 'u'.
    else if (color[vecino] == color[u]) {
        // El grafo NO es bipartito (hay dos nodos conectados con el mismo color).
        return false;
    }
    // CONDICION 3: Vecino ya coloreado con el color opuesto (todo bien), pasamos al siguiente.
}
```

B. Manejo de Grafos Disconexos

Este ciclo asegura que si el grafo tiene varias "islas" o componentes desconectadas, se verifique la propiedad de bipartición en cada una de ellas independientemente.

```
// 2. ITERACION SOBRE TODOS LOS NODOS
// Recorremos todos los nodos. Esto asegura que visitemos todas las partes
// del grafo, incluso si estan desconectadas.
for (int inicio = 0; inicio < V; inicio++) {

    // Si el nodo 'inicio' no tiene color (-1), iniciamos un nuevo chequeo en esta parte.
    if (color[inicio] == -1) {
```

g) 2. Grafo Bipartito (Multiplicación de Matrices)

Tipo de Grafo: No Dirigido, No Ponderado (se ignoran pesos).

Análisis de Complejidad (Big-O)

- **Tiempo:** O(V^4)
  - La multiplicación de dos matrices cuadradas de tamaño V X V tiene un costo de O(V^3). Para detectar ciclos de longitud impar, el algoritmo eleva la matriz de adyacencia a potencias impares (3, 5,..., V). Dado que se realizan aproximadamente $V/2$ iteraciones y en cada una se ejecuta la multiplicación matricial, la complejidad total escala a O(V X V^3) = O(V^4). Es muy ineficiente comparado con BFS (O(V+E)), por lo que se limita a grafos pequeños (MAX_NODES = 10).
- **Espacio:** O(V^2)
  - Se requiere almacenar la matriz de adyacencia y las matrices temporales resultantes de la multiplicación, ocupando memoria cuadrática.

Explicación del Funcionamiento

Este enfoque algebraico se basa en el teorema que establece que un grafo es bipartito si y solo si **no contiene ciclos de longitud impar**. El elemento (i, j) de la matriz de adyacencia elevada a la potencia k (A^k) representa el número de caminos de longitud k entre el nodo i y el nodo j. Si en alguna potencia impar k (donde k >= 3), algún elemento de la diagonal principal (i, i) es mayor que 0, significa que existe un camino de longitud impar que empieza y termina en i (un ciclo impar), lo que demuestra que el grafo no es bipartito.

Fragmentos de Código Relevantes

A. Multiplicación de Matrices (O(V^3))

Este es el motor computacional del algoritmo. Calcula los caminos combinando dos matrices.

```
// 1. FUNCION PARA MULTIPLICAR DOS MATRICES Cuadradas (A * B)
// En terminos de grafos, si A es el 'mapa de caminos de longitud 1',
// esta funcion combina dos mapas (A y B) para crear un mapa de caminos mas largos.
vector< vector<int> > multiplicar_matrices(const vector< vector<int> >& A,
                                           const vector< vector<int> >& B,
                                           int V)
{
    // C es la matriz (el nuevo mapa de caminos) donde guardaremos el resultado.
    vector< vector<int> > C(V, vector<int>(V, 0));

    // Estos tres ciclos son la forma estandar en que se multiplican las matrices.
    // Es como revisar todas las combinaciones posibles de caminos intermedios.
    for (int i = 0; i < V; ++i) { // 'i' es el nodo de INICIO del camino.
        for (int j = 0; j < V; ++j) { // 'j' es el nodo de LLEGADA del camino.
            for (int k = 0; k < V; ++k) { // 'k' es el nodo INTERMEDIO por el que pasamos.
                // Aqui acumulamos la cantidad de formas de ir de 'i' a 'j' pasando por 'k'.
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return C; // Devolvemos el nuevo mapa de caminos.
}
```

B. Detección de Ciclos Impares en la Diagonal

Aquí se iteran las potencias impares y se verifica la diagonal principal. Si potencia_actual[i][i] > 0, se confirma el ciclo impar.

```
// Bucle para buscar CICLOS IMPARES: 3 (triangulo), 5, 7, y asi hasta el numero de nodos.
for (int k = 3; k <= V; k += 2) {

    // Para llegar a la potencia impar 'A^k', necesitamos multiplicar dos veces la matriz.

    // Multiplicacion 1: Nos lleva a una longitud PAR (k-1). No la revisamos.
    // Ejemplo: Si buscamos A^3, aqui calculamos A^2.
    potencia_actual = multiplicar_matrices(potencia_actual, matriz_adyacencia, V);

    // Multiplicacion 2: Nos lleva a la longitud IMPAR 'k'. Esta es la que revisamos.
    // Ejemplo: Si buscamos A^3, aqui finalmente calculamos A^3.
    potencia_actual = multiplicar_matrices(potencia_actual, matriz_adyacencia, V);

    cout << " -> Verificando ciclos de longitud IMPAR: " << k << endl;

    // Verificamos la DIAGONAL PRINCIPAL (donde la fila 'i' es igual a la columna 'i').
    // Si (A^k)[i][i] es mayor que 0, significa que existe un camino de longitud 'k'
    // que va del nodo 'i' a si mismo. ¡Esto es un ciclo!
    for (int i = 0; i < V; ++i) {
        if (potencia_actual[i][i] > 0) {
            // Si encontramos un ciclo de longitud impar, ¡el grafo NO es bipartito!
            cout << "[CONFLICTO]: Se encontro un ciclo de longitud impar " << k << " que pasa por el nodo " << (i + 1) << "." << endl;
            return false; // Salimos de la funcion, el resultado es NO bipartito.
        }
    }
}
```

g) 3. Grafo Bipartito (con DFS)

Tipo de Grafo: No Dirigido, No Ponderado (se ignoran pesos).

Análisis de Complejidad (Big-O)

- **Tiempo:** O(V + E)
  - El algoritmo emplea una Búsqueda en Profundidad (DFS). En el peor de los casos, la función recursiva se invoca una vez por cada vértice (V) y el ciclo que revisa los vecinos se ejecuta dos veces por cada arista (E) en total. Por lo tanto, el tiempo de ejecución es lineal.
- **Espacio:** O(V + E)
  - La estructura de datos dominante es la lista de adyacencia, que consume espacio proporcional al tamaño del grafo. Además, el vector de colores y la pila de recursión del sistema consumen memoria lineal (O(V)).

Explicación del Funcionamiento

Este enfoque verifica la propiedad de bipartición coloreando el grafo con dos colores (0 y 1). Utiliza DFS (recursión) para explorar el grafo. Comienza asignando un color a un nodo y luego intenta colorear a todos sus vecinos con el color opuesto. Si en cualquier punto de la exploración profunda encuentra un vecino que ya tiene asignado el mismo color que el nodo actual, se confirma una contradicción (un ciclo de longitud impar) y el algoritmo retorna false. El proceso se repite para todos los nodos no visitados para cubrir grafos disconexos.

Fragmentos de Código Relevantes

A. Verificación Recursiva y Detección de Conflictos

Este es el núcleo lógico. Intenta colorear recursivamente. Si el vecino no tiene color, profundiza. Si ya tiene color y es igual al actual, detecta el conflicto.

```
// 2. RECORRER VECINOS Y COMPROBAR LA REGLA BIPARTITA
for (size_t i = 0; i < grafo[u].size(); ++i) {

    int vecino = grafo[u][i];

    // Caso A: El vecino NO ha sido coloreado.
    if (color[vecino] == -1) {

        // Asignamos el color OPUESTO (1 - color_actual) y nos sumergimos recursivamente.
        if (!DFS_Bipartito(vecino, 1 - color_actual, grafo, color)) {
            return false; // Conflicto detectado en la rama profunda.
        }
    }
    // Caso B: El vecino YA esta coloreado con el MISMO color.
    else if (color[vecino] == color_actual) {

        // ¡VIOLACION DE LA REGLA! El grafo no es bipartito.
        cout << "\n[CONFLICTO ENCONTRADO]: Los nodos " << u << " y " << vecino
             << " estan conectados y ambos pertenecen al color " << color_actual
             << " (" << (color_actual == 0 ? "Grupo A" : "Grupo B") << ")." << endl;
        return false;
    }
    // Caso C: El vecino YA esta coloreado con el color opuesto. (Esta bien, continuamos).
}
```

B. Iteración para Grafos Disconexos

Garantiza que se verifiquen todas las componentes del grafo, no solo la conectada al nodo 0.

```
// 5. ITERAR SOBRE TODOS LOS NODOS (Maneja grafos disconexos)
for (int i = 0; i < V; ++i) {
    if (color[i] == -1) {
        if (!DFS_Bipartito(i, 0, grafo_adyacencia, color)) {
            es_bipartito = false;
            break;
        }
    }
}
```

h) 1. Emparejamiento Máximo en Grafos Generales: Algoritmo de Edmonds (Blossom)

Tipo de Grafo: General (Dirigido/No Dirigido, usualmente No Dirigido para Matching), No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(V^3)
  - El algoritmo busca caminos aumentantes iterativamente. En el peor de los casos, realiza V/2 iteraciones (una por cada emparejamiento posible). En cada iteración, ejecuta una búsqueda (BFS) que toma O(E). Sin embargo, la operación crítica es la "contracción de blossoms" (ciclos impares). Cuando se detecta un ciclo impar, se debe encontrar el Ancestro Común Más Bajo (LCA) y contraer los nodos, lo que puede llevar O(V) por contracción. Dado que pueden ocurrir múltiples contracciones anidadas, la complejidad efectiva de una búsqueda se eleva a O(V^2), resultando en un total cúbico O(V^3).
- **Espacio:** O(V + E)
  - Se utiliza una lista de adyacencia para el grafo (V + E). Además, se requieren varios arreglos auxiliares de tamaño V para gestionar el estado del algoritmo: matchNode (parejas), parentNode (árbol de búsqueda), baseNode (identificador de blossom) y banderas de visitados.

Explicación del Funcionamiento

Este algoritmo resuelve el problema del Matching Máximo en grafos generales, superando la limitación de los grafos bipartitos. La dificultad principal en grafos generales es la existencia de "ciclos impares". Cuando el algoritmo explora caminos alternantes y encuentra un ciclo impar, este ciclo se denomina "Blossom".

El algoritmo de Edmonds identifica estos ciclos y los "contrae" virtualmente en un solo super-nodo. Esto permite que la búsqueda continúe como si el ciclo fuera un solo vértice. Una vez encontrado un camino aumentante a través del grafo contraído, el algoritmo "expande" los blossoms y ajusta las aristas para obtener el emparejamiento válido en el grafo original.

Fragmentos de Código Relevantes

A. Cálculo del LCA y Contracción (Manejo de Blossoms)

Este es el componente distintivo de Edmonds. Identifica la base común del ciclo impar para poder contraerlo.

```
// ================================================================
// BUSCAR CAMINO AUMENTANTE (BFS modificado)
// ================================================================
int findPath(int start) {
    for (int i = 0; i < n; ++i) used[i] = false;
    for (int i = 0; i < n; ++i) parentNode[i] = -1;
    for (int i = 0; i < n; ++i) baseNode[i] = i;

    while (!q.empty()) q.pop();

    q.push(start);
    used[start] = true;

    while (!q.empty()) {
        int v = q.front(); q.pop();

        for (int idx = 0; idx < (int)adj[v].size(); ++idx) {
            int u = adj[v][idx];

            if (baseNode[v] == baseNode[u] || matchNode[v] == u)
                continue;

            if (u == start ||
                (matchNode[u] != -1 && parentNode[matchNode[u]] != -1)) {

                int b = lca(v, u);
                for (int i = 0; i < n; ++i) blossom[i] = false;
                markPath(v, b, u);
                markPath(u, b, v);
```

```
                for (int i = 0; i < n; ++i) {
                    if (blossom[baseNode[i]]) {
                        baseNode[i] = b;
                        if (!used[i]) {
                            used[i] = true;
                            q.push(i);
                        }
                    }
                }
            }
            else if (parentNode[u] == -1) {
                parentNode[u] = v;

                if (matchNode[u] == -1)
                    return u;

                int next = matchNode[u];
                used[next] = true;
                q.push(next);
            }
        }
    }

    return -1;
}
```

B. Detección de Ciclos y Modificación del Grafo

Dentro del BFS (findPath), cuando se encuentra una arista hacia un nodo ya visitado que forma un ciclo impar, se llama a la contracción.

```
if (u == start ||
    (matchNode[u] != -1 && parentNode[matchNode[u]] != -1)) {

    int b = lca(v, u);
    for (int i = 0; i < n; ++i) blossom[i] = false;
    markPath(v, b, u);
    markPath(u, b, v);

    for (int i = 0; i < n; ++i) {
        if (blossom[baseNode[i]]) {
            baseNode[i] = b;
            if (!used[i]) {
                used[i] = true;
                q.push(i);
            }
        }
    }
}
else if (parentNode[u] == -1) {
    parentNode[u] = v;

    if (matchNode[u] == -1)
        return u;

    int next = matchNode[u];
    used[next] = true;
    q.push(next);
}
```

h) 2. Pareo (Matching) Maximal: Algoritmo Greedy

Tipo de Grafo: No Dirigido, Ponderado/No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(E)
  - El algoritmo recorre linealmente la lista de aristas (E) tal como fueron ingresadas. Para cada arista, realiza una verificación de tiempo constante (O(1)) para ver si sus nodos extremos ya han sido emparejados. No hay ordenamiento ni retroceso, por lo que la complejidad es estrictamente lineal respecto al número de aristas.
- **Espacio:** O(V + E)
  - Se requiere almacenar la lista de aristas (E) y un arreglo de banderas usado de tamaño V para marcar los vértices ya emparejados.

Explicación del Funcionamiento

Este es un algoritmo heurístico voraz que construye un pareo maximal (no extensible). Itera sobre las aristas disponibles en el orden de entrada. Si una arista conecta dos nodos que aún no forman parte de ninguna pareja, la selecciona inmediatamente para el conjunto de pareo y marca ambos nodos como "ocupados". Si alguno de los nodos ya está ocupado, la arista se descarta. El resultado es un conjunto de aristas independientes al que no se le puede añadir ninguna otra arista del grafo original, aunque no garantiza ser el pareo de cardinalidad máxima.

Fragmentos de Código Relevantes

A. Selección Greedy

Este ciclo contiene toda la lógica. Verifica la disponibilidad de u y v; si ambos están libres, "congela" la pareja y bloquea los nodos para futuras aristas.

```
for (int i = 0; i < m; i++) {
    int u = aristas[i].u;
    int v = aristas[i].v;

    if (!usado[u] && !usado[v]) {

        matching.push_back(aristas[i]);
        usado[u] = true;
        usado[v] = true;

        cout << " -> [ACEPTADA] Arista (" << u << " , " << v
             << ") Peso: " << aristas[i].peso << ". Nodos bloqueados.\n";
    }
    else {
        cout << " -> [RECHAZADA] Arista (" << u << " , " << v
             << ") -> Conflicto con nodo(s) ya emparejado(s).\n";
    }
}
```

h) 3. Pareos perfectos y maximales: Algoritmo de Hopcroft-Karp

Tipo de Grafo: Bipartito, No Dirigido, No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(E \sqrt{V})
  - Este algoritmo mejora significativamente el enfoque tradicional de caminos aumentantes (O(VE)). Funciona en fases; en cada fase, utiliza una búsqueda en anchura (BFS) para encontrar el conjunto máximo de caminos aumentantes *más cortos* disjuntos, y luego una búsqueda en profundidad (DFS) para aplicarlos. Se demuestra matemáticamente que el número de fases necesarias está acotado por \sqrt{V), y cada fase recorre las aristas (E), resultando en la complejidad mencionada.
- **Espacio:** O(V + E)
  - Se requiere almacenar el grafo mediante listas de adyacencia (V+E). Adicionalmente, se utilizan vectores auxiliares de tamaño V para almacenar las distancias (dist), los emparejamientos (pairU, pairV) y la cola del BFS.

Explicación del Funcionamiento

El algoritmo de Hopcroft-Karp busca maximizar el número de aristas emparejadas en un grafo bipartito. A diferencia de algoritmos más simples que buscan un solo camino aumentante por iteración, este algoritmo busca **múltiples** caminos aumentantes simultáneamente en cada fase.

1. **Fase BFS:** Divide el grafo en capas basándose en la distancia desde los nodos libres (sin pareja). Esto permite identificar los caminos aumentantes más cortos disponibles.
2. Fase DFS: Utiliza la información de capas del BFS para buscar y aumentar el emparejamiento a lo largo de estos caminos cortos.

   Este ciclo se repite hasta que no sea posible encontrar más caminos aumentantes.

Fragmentos de Código Relevantes

A. Fase BFS (Cálculo de Distancias y Capas)

Este bloque es crucial para la eficiencia del algoritmo. Construye las capas de distancia (dist) solo para los caminos que pueden llevar a un aumento del emparejamiento, deteniéndose cuando alcanza el nodo nulo (NIL) con la distancia mínima encontrada.

```
// Si la distancia al nodo 'u' no es infinita (es decir, es un nodo alcanzable en este nivel).
if (dist[u] < dist[NIL]) {
    // Recorre todos los vecinos (conexiones) del nodo 'u'.
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        // Verifica la pareja del vecino: si la pareja del vecino 'v' (pairV[v])
        // tiene distancia infinita, significa que su camino no ha sido explorado todavía.
        if (dist[pairV[v]] == INF) {
            // Actualiza la distancia: la pareja de 'v' esta un paso mas alla que 'u'.
            dist[pairV[v]] = dist[u] + 1;
            // Agrega la pareja de 'v' a la cola.
            q.push(pairV[v]);
```

B. Fase DFS (Aumento del Emparejamiento)

Utiliza las distancias calculadas por el BFS para guiar la recursión. Solo avanza si el siguiente nodo en el camino cumple con ser exactamente un paso más distante (dist[pairV[v]] == dist[u] + 1), garantizando que se usen los caminos más cortos.

```
// Condicion clave: Solo sigue el camino si es el camino mas corto (dist[pairV[v]] == dist[u] + 1)
// Y si la llamada recursiva (dfs) tambien encuentra una ruta hasta el final.
if (dist[pairV[v]] == dist[u] + 1 &&
    dfs(pairV[v], adj, dist, pairU, pairV)) {

    // ¡Exito! Se encontro y se uso un camino aumentante.
    // Se actualizan las parejas:
    pairU[u] = v; // 'u' se empareja con 'v'.
    pairV[v] = u; // 'v' se empareja con 'u'.
    return true; // Devuelve verdadero para que el camino anterior sepa que tuvo exito.
}
```

i) 3. Pareos de peso máximo en bipartitos: Algoritmo Húngaro (Kuhn-Munkres)

Tipo de Grafo: Bipartito, No Dirigido, Ponderado (Asignación Óptima).

Análisis de Complejidad (Big-O)

- **Tiempo:** O(V^3)
  - Esta implementación del algoritmo Húngaro tiene una complejidad cúbica. El ciclo exterior itera V veces (para emparejar cada fila). Dentro de este, la búsqueda del camino aumentante y la actualización de potenciales (etiquetas u y v) toman O(V^2) gracias al uso del arreglo auxiliar minv, que evita recalcular mínimos desde cero en cada paso.
- **Espacio:** O(V^2)
  - Se requiere una matriz de costos de tamaño V x V. Adicionalmente, se utilizan varios arreglos lineales de tamaño V (u, v, p, way, minv, usado), pero la matriz domina el consumo de memoria.

Explicación del Funcionamiento

El algoritmo Húngaro resuelve el problema de asignación (matching perfecto de peso máximo/mínimo en grafos bipartitos completos). Utiliza un enfoque basado en etiquetas (potenciales) para los nodos.

1. **Transformación:** Como el algoritmo nativo minimiza costos, para maximizar pesos transformamos la matriz restando cada peso del valor máximo posible (C_{ij}' = M - C_{ij}).
2. **Aumento:** Si no encuentra un camino, ajusta las etiquetas (usando el valor delta) para introducir nuevas aristas candidatas sin violar las restricciones, repitiendo hasta completar el pareo.

Fragmentos de Código Relevantes

A. Transformación para Maximización

Este paso es crucial para adaptar el algoritmo estándar de minimización al problema de "Peso Máximo" solicitado.

```
// Se resta cada peso de la matriz del peso mas grande encontrado (maxValor).
// Ejemplo: Si el maxValor es 10. Un peso de 8 se convierte en 2 (10-8).
// Ahora, el camino mas "barato" (2) en la matriz transformada es el camino mas "caro" (8) en la matriz original.
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        costo[i][j] = maxValor - costo[i][j];
    }
}
```

B. Ajuste de Etiquetas (Delta)

Este bloque representa la actualización de potenciales para desbloquear nuevos caminos cuando la búsqueda actual se estanca.

```
// Esta parte clave ajusta las etiquetas (u y v) para que el a
for (int j = 0; j <= n; j++) {
    if (usado[j]) {
        u[p[j]] += delta;
        v[j] -= delta;
    } else {
        minv[j] -= delta;
    }
}
```

i) 1. Emparejamiento Máximo en Grafos Generales: Algoritmo de Edmonds (Blossom)

Tipo de Grafo: General (Dirigido/No Dirigido, usualmente No Dirigido para Matching), No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(E \sqrt{V}) para versiones optimizadas o O(V^3) en implementaciones estándar.
  - El algoritmo busca caminos aumentantes. En grafos generales, la presencia de "blossoms" (ciclos de longitud impar) complica la búsqueda. El algoritmo debe identificar estos ciclos, contraerlos en un "super-nodo" y continuar la búsqueda. La contracción y expansión de blossoms añade una capa de complejidad sobre el BFS/DFS estándar. En la implementación más común (como la presentada), se realizan iteraciones de búsqueda que pueden tomar O(E), y en el peor de los casos se contraen O(V) blossoms, resultando en una complejidad cúbica.
- **Espacio:** O(V + E)
  - Se almacena el grafo (listas de adyacencia) y varios arreglos auxiliares de tamaño lineal (V) para mantener el estado del emparejamiento (match), el árbol de búsqueda (parent), las bases de los blossoms (base) y las etiquetas (label).

Explicación del Funcionamiento

Este algoritmo resuelve el problema del *Matching* Máximo en cualquier tipo de grafo, no solo bipartitos. La clave es el manejo de los **ciclos impares** (blossoms).

1. **Búsqueda de Caminos Aumentantes:** Inicia un BFS desde nodos libres para encontrar un camino hacia otro nodo libre que alterne entre aristas emparejadas y no emparejadas.

2. **Contracción de Blossoms:** Si el BFS encuentra una arista que conecta dos nodos del mismo nivel en el árbol de búsqueda (formando un ciclo impar), identifica este ciclo como un "blossom". El algoritmo "contrae" virtualmente todos los nodos del ciclo en un solo nodo base. Esto simplifica el grafo temporalmente, permitiendo que la búsqueda continúe a través del ciclo como si fuera un vértice normal.
3. **Expansión y Aumento:** Si se encuentra un camino aumentante en el grafo contraído, se "expande" de vuelta al grafo original, ajustando el emparejamiento dentro de los blossoms para mantener la validez, y se invierte el camino para aumentar el tamaño del matching.

Fragmentos de Código Relevantes

A. Detección y Contracción de Blossoms

Este es el corazón del algoritmo. Si el vecino v ya fue visitado y tiene la misma etiqueta (label[v] == 1) que el nodo actual u (ambos son "pares" o "impares" desde la raíz), se ha encontrado un ciclo impar. Se calcula el Ancestro Común Más Bajo (LCA) y se contrae el ciclo.

```
} else if (label[v] == 1) {
    // Encontramos un blossom
    int lca = find_lca(u, v);
tra una arista que conecta dos nodos con la misma etiqueta (1), se
    if (lca != 0) {
        contract_blossom(u, v, lca);
a funcion para encoger el Blossom.
    }
}
```

B. Búsqueda del Ancestro Común (LCA)

Esta función auxiliar es vital para identificar la "base" del blossom, es decir, el punto donde el ciclo se conecta con el resto del árbol de búsqueda alternante.

```
int find_lca(int u, int v) {
'lca' se usa para encontrar el punto donde se encuentran los caminos que salen de 'u' y 'v'.
Si los caminos se encuentran y estan etiquetados de forma similar, significa que se encontro un Blos
    vector<bool> visited(V + 1, false);

    // Alternamos entre u y v hasta encontrar un nodo visitado
    while (true) {
        u = get_base(u);
        if (u != 0) {
            if (visited[u]) return u;
            visited[u] = true;
Si el nodo 'u' ya esta emparejado, subimos al padre de su pareja para seguir buscando.
            if (match[u] != -1) u = parent[match[u]];
            else u = 0;
        }

        swap(u, v); // Alternamos para buscar desde ambos lados
    }
}
```

i) 4. Pareos maximales en no bipartitos: Algoritmo Greedy

Tipo de Grafo: General (No necesariamente Bipartito), No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(E)
  - El algoritmo implementa un enfoque voraz (greedy) simple. Itera sobre la lista de todas las aristas (todasAristas) exactamente una vez. Para cada arista, realiza operaciones de tiempo constante (O(1)) para verificar si los nodos extremos están cubiertos y marcarlos si es necesario. No hay ciclos anidados ni retrocesos, por lo que la complejidad es lineal respecto al número de aristas.
- **Espacio:** O(V+E)
  - Se utiliza un vector para almacenar la lista de aristas (O(E)). Adicionalmente, se emplea un arreglo booleano cubierto de tamaño V para rastrear el estado de cada nodo.

Explicación del Funcionamiento

Este algoritmo construye un **Pareo Maximal** (no confundir con Máximo) en cualquier tipo de grafo. Un pareo es maximal si no se puede añadir ninguna otra arista al conjunto sin romper la propiedad de pareo (es decir, sin compartir vértices). El funcionamiento es directo: procesa las aristas en el orden en que fueron ingresadas o almacenadas. Si encuentra una arista cuyos dos extremos están libres (no han sido cubiertos por una arista seleccionada previamente), la añade al pareo y marca ambos nodos como ocupados. Si alguno de los extremos ya está ocupado, la arista se descarta.

Fragmentos de Código Relevantes

A. Selección Voraz (Greedy) Este ciclo es el único paso del algoritmo. Verifica la disponibilidad de los nodos u y v. Si ambos están libres (!cubierto), se emparejan inmediatamente y quedan indisponibles para futuras aristas.

```
int greedyMaximalMatching() {
    matchingResultante.clear();

    for (int i = 0; i <= nNodos; ++i)
        cubierto[i] = false;

    cout << "\n[PROCESO] Analizando aristas en orden codicioso...\n";
    for (size_t i = 0; i < todasAristas.size(); ++i) {

        int u = todasAristas[i].first;
        int v = todasAristas[i].second;

        if (!cubierto[u] && !cubierto[v]) {
            matchingResultante.push_back(todasAristas[i]);
            cubierto[u] = true;
            cubierto[v] = true;
            cout << " -> Emparejado: (" << u << " -- " << v << ")\n";
        } else {
            cout << " -> Ignorado: (" << u << " -- " << v << ") - Nodos ya cubiertos.\n";
        }
    }

    return matchingResultante.size();
}
```

B. Verificación de Perfección Aunque el algoritmo solo garantiza un pareo maximal, este bloque verifica si, por casualidad, el resultado también es un pareo perfecto (cubre todos los nodos).

```
int nodosCubiertos = 0;
for (int i = 1; i <= nNodos; ++i)
    if (cubierto[i]) nodosCubiertos++;

cout << "\nVERIFICACION DE PERFECCION:\n";
cout << "  Nodos cubiertos: " << nodosCubiertos << " de " << nNodos << endl;

if (nodosCubiertos == nNodos)
    cout << "  [PERFECTO] El pareo SI cubre todos los nodos.\n";
else
    cout << "  [NO PERFECTO] Faltan " << (nNodos - nodosCubiertos) << " nodos.\n";
```

i) 5. Pareo Maximal: Heurística Aleatoria (Random Greedy)

Tipo de Grafo: General (No Dirigido), No Ponderado.

Análisis de Complejidad (Big-O)

- **Tiempo:** O(E)
  - El algoritmo consta de dos pasos principales: primero, mezclar aleatoriamente el vector de aristas, lo cual se realiza en tiempo lineal O(E) usando random_shuffle. Segundo, recorrer linealmente la lista mezclada para seleccionar las aristas, realizando operaciones de tiempo constante (O(1)) por cada una. La complejidad total se mantiene lineal respecto al número de aristas.
- **Espacio:** O(V + E)
  - Se utiliza un vector para almacenar todas las aristas del grafo (E). Adicionalmente, se emplea un arreglo booleano cubierto de tamaño V para rastrear el estado de cada nodo.

Explicación del Funcionamiento

Este algoritmo construye un **Pareo Maximal**. A diferencia del algoritmo Greedy estándar determinista (que depende del orden de entrada de los datos), esta versión introduce un paso de pre-procesamiento que reordena las aristas al azar.

1. **Aleatorización:** Se "barajan" las aristas para evitar sesgos provocados por un orden de entrada específico que podría llevar a un pareo muy pequeño.
2. Selección Voraz: Itera sobre la lista mezclada. Si encuentra una arista cuyos dos extremos están libres, la añade al pareo y marca los nodos como ocupados.

Aunque sigue sin garantizar un Pareo Máximo (el más grande posible), la aleatorización suele producir resultados promedio mejores o distintos en ejecuciones consecutivas comparado con el enfoque estático.

Fragmentos de Código Relevantes

A. Aleatorización del Orden

Esta es la única diferencia con el algoritmo Greedy estándar. Se altera el orden de procesamiento para explorar diferentes configuraciones del pareo maximal.

```
cout << "\n[PROCESO] Barajando el orden de las aristas aleatoriamente...\n";

// Mezclar el orden de las aristas
random_shuffle(todasAristas.begin(), todasAristas.end());

cout << "[PROCESO] Analizando aristas en orden aleatorio Codicioso...\n";
```

B. Lógica Codiciosa (Greedy)

Una vez mezcladas, la lógica es idéntica al algoritmo voraz: el primero que llega y está libre, se queda.

```
// Recorrer todas las aristas
for (size_t i = 0; i < todasAristas.size(); ++i) {
    int u = todasAristas[i].first;
    int v = todasAristas[i].second;

    // Si ambos nodos están libres, aceptar la arista
    if (!cubierto[u] && !cubierto[v]) {
        matchingResultante.push_back(todasAristas[i]);
        cubierto[u] = true;
        cubierto[v] = true;

        cout << " -> Emparejado: (" << u << " -- " << v << ")\n";
    }
}
```

ANÁLISIS Y DISCUSIONES

El proyecto se centró en la implementación exhaustiva de algoritmos fundamentales de teoría de grafos utilizando C++. La elección de estructuras de datos, particularmente la matriz de adyacencia para la mayoría de las implementaciones, influyó directamente en las complejidades temporales y espaciales reportadas. Aunque esta representación facilita ciertas operaciones (como verificar la existencia de una arista en O(1)), incrementa el costo en grafos dispersos para algoritmos de recorrido y componentes conexas.

La metodología Scrum permitió una organización eficiente del trabajo en un tiempo limitado (una semana), con claridad en la distribución de responsabilidades y seguimiento continuo del progreso mediante GitHub Projects. La integración de ceremonias ágiles (planificación, dailies, revisión y retrospectiva) contribuyó a mantener el enfoque y resolver impedimentos oportunamente.

CONCLUSIONES

El proyecto "Repositorio acerca de algoritmos para grafos" se completó exitosamente en el plazo establecido de una semana. Se implementaron, probaron y documentaron más de 20 algoritmos diferentes de teoría de grafos, abarcando desde representaciones básicas hasta algoritmos avanzados de pareo y componentes fuertemente conexas.

La aplicación de la metodología Scrum demostró ser efectiva para gestionar proyectos de desarrollo de software con múltiples integrantes y tareas técnicas complejas. La combinación de roles definidos (Scrum Master, Product Owner, Developers), ceremonias regulares y herramientas de seguimiento (GitHub Projects) facilitó la coordinación y el cumplimiento de los objetivos del sprint.

El análisis de complejidad realizado para cada algoritmo proporciona una comprensión clara de las limitaciones y aplicabilidad de cada implementación, lo cual es fundamental para tomar decisiones informadas sobre qué algoritmo utilizar en diferentes contextos prácticos.

REFERENCIAS

Aprende con IA. (2025, 27 octubre). Entendiendo grafos bipartitos [Vídeo]. YouTube. https://www.youtube.com/watch?v=2bWkZX6WrBM

UCAM Universidad Católica de Murcia. (2016, 21 abril). Matemática Discreta - Grafo bipartido - Jesús Soto [Vídeo]. YouTube. https://www.youtube.com/watch?v=oxslxaHiQEI

Universitat Politècnica de València - UPV. (2011, 22 septiembre). Problema de emparejamientos: Emparejamiento máximo en un grafo bipartido a partir de u | 25/42 | UPV [Vídeo]. YouTube. https://www.youtube.com/watch?v=ArggJZU0kkY

Usha's EduVids. (2024, 28 junio). Graph Matching,Maximal Matching,Maximum matching,Perfect Matching [Vídeo]. YouTube. https://www.youtube.com/watch?v=lGdo77LBNXQ

OptWhiz. (2022, 1 noviembre). Can we assign everyone a job? (maximum matchings) | Bipartite Matchings [Vídeo]. YouTube. https://www.youtube.com/watch?v=ELcgI_C1mNM

- Varun Sir. (s.f.). L-4.15: BFS & DFS | Breadth First Search | Depth First Search [Video]. YouTube. https://www.youtube.com/watch?v=N2P7w22tN9c

- WilliamFiset. (2020, April 22). Tarjan's Strongly Connected Component (SCC) Algorithm (UPDATED) | Graph Theory [Video]. YouTube. https://www.youtube.com/watch?v=yrbzNYcdhQI&t=97s

The code bit (October,2025). Gabow's Algorithm Explained [Video]. YouTube. Gabow's Algorithm Explained

- Potato Coders. (2020, December 03). Union Find in 5 minutes – Data Structures & Algorithms [Video]. YouTube. https://www.youtube.com/watch?v=ymxPZk7TesQ

HeadEasy.(2023). Kosaraju's Algorithm | Strongly Connected Components [Video]. YouTube. Kosaraju's Algorithm | Strongly Connected Components – YouTube

Juan Villalpando. (2021). Método Kruskal y Prim; Árbol Recubridor Mínimo y Máximo [Video]. YouTube. https://www.youtube.com/watch?v=J_uZWi1jyKI

Balvin, J. (2022). Árbol de Expansión Mínima (Prim + Kruskal) [Video]. YouTube.

https://www.youtube.com/watch?v=srGy1bSPcCQ

Rodríguez, F. (2020). Grafos: Árbol parcial mínimo con algoritmo de Prim [Video]. YouTube.

https://www.youtube.com/watch?v=6pPSPYUbTIw

Estudiante Digital. (2021). Árbol de Expansión Mínima – Algoritmo de Kruskal (Explicación paso a paso) [Video]. YouTube.

https://www.youtube.com/watch?v=ITCDUJw_4GM