ESCUELA SUPERIOR POLITECNICA DE CHIMBORAZO

FACULTAD DE INFORMATICA Y ELECTRONICA

CARRERA DE SOFTWARE

TEMA

ORDENACIÓN TOPOLÓGICA

ESTUDIANTES **CÓDIGOS**

[redacted] [redacted]

ASIGNATURA

ESTRUCTURAS DE DATOS

PROFESOR

[redacted]

FECHA DE ENTREGA

08/07/2024

RIOBAMBA - ECUADOR

Introducción

En el ámbito de la teoría de grafos la ordenación topológica es una herramienta fundamental para organizar y analizar estructuras complejas de datos que presentan relaciones de precedencia la ordenación topológica se aplica a grafos dirigidos acíclicos (DAG, por sus siglas en inglés) y proporciona un orden lineal de los vértices tal que, para cualquier arista dirigida de un vértice u a un vértice v, u aparece antes que v en el orden resultante. Este proceso es esencial para resolver problemas en diversos campos, como la planificación de tareas, la compilación de programas y el análisis de dependencias.

Además, la ordenación topológica no solo ayuda a estructurar y organizar la información dentro de un grafo, sino que también identifica inconsistencias o ciclos, que pueden ser críticos para la integridad y funcionalidad de los sistemas representados. Por ejemplo, en la gestión de proyectos, la ordenación topológica asegura que todas las dependencias entre tareas se respeten, optimizando el flujo de trabajo y la utilización de recursos. En el contexto de compiladores, garantiza que los módulos se compilen en el orden correcto, evitando errores de dependencia.

Dos de los algoritmos más destacados para realizar la ordenación topológica son el Algoritmo de Kahn y el Algoritmo de Búsqueda en Profundidad (DFS) el Algoritmo de Kahn utiliza una cola de vértices con grado de entrada cero y procesa secuencialmente los vértices, mientras que el Algoritmo de DFS explora el grafo en profundidad, utilizando una estrategia recursiva o una pila explícita. Ambos enfoques presentan características y eficiencias que los hacen adecuados para diferentes tipos de grafos y contextos de aplicación.

Desarrollo

Definición de Grafos

En primer lugar, en el ámbito de la teoría de grafos, se estudian estructuras matemáticas que modelan relaciones entre objetos además un grafo G se define formalmente como un par *G = (V, E),* donde V es un conjunto de vértices (nodos) y E es un conjunto de aristas (conexiones entre pares de vértices). Los grafos permiten representar una amplia gama de relaciones en diversas disciplinas, desde la informática hasta la biología y la logística, proporcionando un marco abstracto y poderoso para el análisis de redes y sistemas complejos.

Tipos de Grafos

Por otro lado, existen dos tipos principales de grafos:

- Grafos No Dirigidos: Las aristas no tienen dirección asociada y representan relaciones simétricas entre los vértices por ejemplo en un grafo no dirigido si hay una arista entre los vértices v y w se puede atravesar en ambas direcciones sin distinción.
- Grafos Dirigidos: En cambio las aristas tienen dirección, indicando una relación direccional de un vértice origen a un vértice destino por ejemplo en un grafo dirigido una arista que va desde v a w no implica necesariamente una arista en la dirección opuesta.

Los grafos dirigidos son útiles para modelar situaciones donde la dirección de la relación entre objetos es relevante como en redes sociales donde se considera la dirección de la amistad o influencia.

Propiedades y Características Básicas de los Grafos

Asimismo, los grafos pueden tener diversas propiedades y características:

- Orden del Grafo: Número de vértices en el grafo.
- Tamaño del Grafo: Número de aristas en el grafo.
- Grado de un Vértice: Número de aristas incidentes en un vértice. Los vértices con grado cero se denominan aislados.
- Conectividad: Propiedad que describe cómo de conectado está un grafo en términos de rutas entre sus vértices. Un grafo puede ser conexo (todos los vértices están conectados) o desconecto (existen componentes desconectados).

Estas propiedades son fundamentales para el análisis estructural y la resolución de problemas prácticos utilizando grafos, como la optimización de redes de transporte o la identificación de comunidades en redes sociales.

Representaciones de Grafos

En cuanto a las representaciones de grafos existen varias formas comunes:

- Matriz de Adyacencia: Una matriz A de tamaño |V|×|V|, donde A[i][j] es 1 si hay una arista entre los vértices i y j, y 0 si no hay arista. En grafos ponderados A[i][j] puede contener el peso de la arista.
- Lista de Adyacencia: Una lista donde cada vértice tiene una lista de vértices adyacentes. Es más eficiente para grafos dispersos donde el número de aristas es mucho menor que el número máximo posible de aristas $|V|^2$

Estas representaciones juegan un papel crucial en la implementación y el rendimiento de algoritmos sobre grafos afectando la complejidad temporal y espacial de las operaciones.

Ordenación Topológica

La ordenación topológica es un concepto fundamental en la teoría de grafos que consiste en un orden lineal de los vértices de un grafo dirigido acíclico (DAG). Este orden respeta las direcciones de las aristas, es decir, si hay una arista dirigida de u a v, entonces u aparece antes que v en la ordenación este concepto es crucial para resolver problemas donde se requiere establecer un orden de precedencia, como en la planificación de tareas o la compilación de programas.

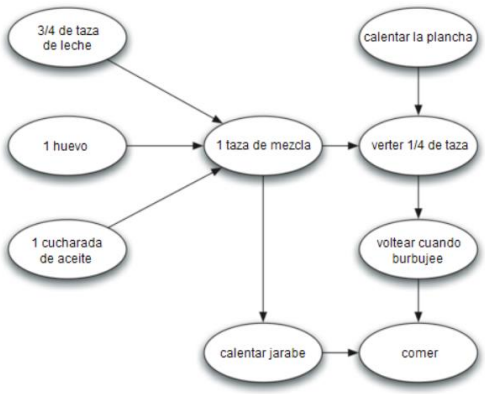

Fig. 1Los pasos para hacer panqueques (pythoned, s/f)

“DEFINICIÓN: Un orden topológico de un digrafo G es un orden lineal de todos los vértices tales que si G contiene un arco (U, V), entonces U aparece antes de V en el ordenamiento. (3.3 ORDENAMIENTOS TOPOLÓGICOS, 2012)”

El ordenamiento topológico es una adaptación simple pero útil de una búsqueda en profundidad. El algoritmo para el ordenamiento topológico es el siguiente:

1. Llamar a bep(g) para algún grafo g. La principal razón por la que queremos invocar a la búsqueda en profundidad es para calcular los tiempos de finalización para cada uno de los vértices.
2. Almacenar los vértices en una lista en orden decreciente según el tiempo de finalización.
3. Devolver la lista ordenada como resultado del ordenamiento topológico.

“La Fig. 2 muestra el bosque de profundidad construido por bep para el grafo de preparación de panqueques mostrado en la Fig. 1. (pythoned, s/f)”

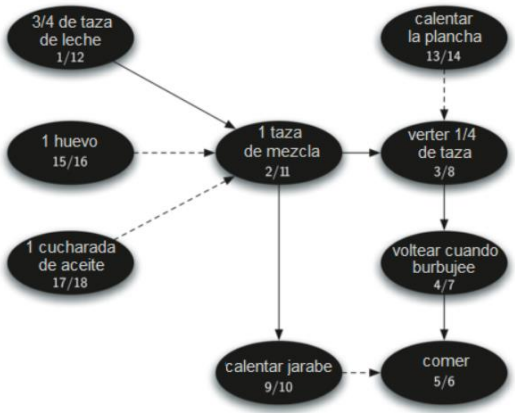

Fig. 2 (pythoned, s/f)

“Fig. 2: Resultado de la búsqueda en profundidad para el grafo de preparación de panqueques

Fig. 2: Resultado de la búsqueda en profundidad para el grafo de preparación de panqueques

Por último, la Fig. 3 muestra los resultados de aplicar el algoritmo de ordenamiento topológico a nuestro grafo. Ahora se ha eliminado toda ambigüedad y sabemos exactamente el orden de los pasos para hacer panqueques. (pythoned, s/f)”

Fig. 3 (pythoned, s/f)

“Fig. 3: Resultado del ordenamiento topológico aplicado a un grafo acíclico dirigido

Fig. 3: Resultado del ordenamiento topológico aplicado a un grafo acíclico dirigido (pythoned, s/f)”

Orden Topológico por Búsqueda en Anchura

En primer lugar, el orden topológico por búsqueda en anchura es un algoritmo utilizado para obtener un orden lineal de los vértices en un grafo dirigido acíclico (DAG). Este método se basa en explorar los vértices en niveles, comenzando por aquellos que no tienen dependencias o cuyas dependencias han sido completamente satisfechas.

Por otro lado, este algoritmo utiliza una estructura de datos como una cola para mantener los vértices que pueden ser procesados en el orden correcto. A medida que se procesan los vértices, se van eliminando las aristas de entrada de sus vértices adyacentes, actualizando así los grados de entrada de esos vértices. Este proceso continúa hasta que todos los vértices han sido procesados.

Además, una ventaja significativa de este enfoque es su capacidad para identificar rápidamente cualquier inconsistencia o ciclo en el grafo, ya que los ciclos en un DAG no permiten un orden topológico válido. Esta característica lo convierte en una herramienta valiosa para verificar la consistencia estructural y lógica de los sistemas modelados por grafos.

“

1. *Crear el arreglo predCount e inicializarlo para que predCount [i] sea el número de predecesores del vértice vi.*
2. *Inicializar la cola, queue, por ejemplo, en todos los vértices v para que predCount [k] sea 0. (Es claro que queue no está vacía porque el grafo no tiene ciclos.)*
3. *En tanto la cola no está vacía, se utiliza while para*

    3.1. Eliminar el elemento u, que está al frente de la cola.

    3.2. Colocar u en la siguiente posición disponible, por ejemplo, topologicalOrder [topIndex], e incrementar topIndex.

    3.3. Para todos los sucesores w inmediatos de u,

        3.3.1. Restar 1 al recuento predecesor de w.

        3.3.2. Si el recuento predecesor de w es 0, agregar w a la cola. (Malik, D, 2013)”

Características de Grafos Acíclicos Dirigidos (DAG)

Por otro lado, los grafos acíclicos dirigidos (DAG) son grafos donde no existen ciclos, es decir, no es posible seguir una secuencia de aristas y regresar al mismo vértice. Esta estructura permite representar relaciones de dependencia o precedencia de manera natural, evitando ambigüedades o conflictos en el ordenamiento de sus vértices.

Importancia en la Resolución de Problemas Prácticos

Asimismo, la ordenación topológica es fundamental en la resolución de problemas prácticos. Por ejemplo, en la planificación de proyectos, permite determinar un orden óptimo de ejecución de tareas basado en sus dependencias mutuas. De igual manera, en la compilación de programas, garantiza que los módulos se compilen en el orden correcto, evitando errores de dependencia.

Algoritmos para la Ordenación Topológica

En cuanto a los algoritmos para la ordenación topológica, existen varios enfoques eficientes:

- Algoritmo de Kahn: Utiliza una cola de vértices con grado de entrada cero y procesa secuencialmente los vértices, eliminando las aristas y actualizando los grados de entrada hasta que todos los vértices son procesados.
- Algoritmo de Depth-First Search (DFS): Utiliza un enfoque recursivo o mediante pila para explorar los vértices en profundidad, marcando los vértices como visitados una vez que todos sus vecinos han sido visitados y agregándolos a una lista en orden inverso.

Estos algoritmos son eficientes y proporcionan soluciones rápidas para la ordenación topológica, adaptándose a diferentes contextos y estructuras de grafos.

Algoritmo de Kahn.

El algoritmo de Kahn es utilizado para realizar el ordenamiento topológico de un grafo dirigido acíclico (DAG, por sus siglas en inglés) utilizando una estrategia basada en grados de entrada de los vértices.

El algoritmo de Kahn funciona identificando vértices sin dependencias de entrada y procesándolos secuencialmente, reduciendo gradualmente el número de dependencias de los vértices restantes. Utiliza una cola para gestionar los vértices con grado de entrada cero y un recuento de grados de entrada para cada vértice.

Paso a Paso:

- Calcular el grado de entrada para cada vértice Fig. 4.
- Inicializar una cola con los vértices que tienen grado de entrada cero Fig. 4.
- Mientras la cola no esté vacía Fig. 5:
  - Extraer un vértice de la cola.
  - Agregar este vértice al orden topológico.
  - Reducir el grado de entrada de sus vértices adyacentes Fig. 5.
  - Si algún vértice adyacente alcanza un grado de entrada cero, agregarlo a la cola Fig. 5.
- Si el orden topológico contiene todos los vértices, se imprime el orden topológico como resultado del algoritmo. Si no, el grafo contiene un ciclo Fig. 6.

```
void ordenamientoTopologicoKahn(GrafoLista &g) {
    int nv = g.getNumVerts();

    // Paso 1: Calcular el grado de entrada para cada vertice
    vector<int> gradosEntrada(nv, 0);
    for (int i = 0; i < nv; ++i) {
        ListaG adyacentes = g.listaAdyacencia(i);
        NodoG *actual = adyacentes.getPrimero();
        while (actual != nullptr) {
            gradosEntrada[actual->getDato()]++;
            actual = actual->getPunt();
        }
    }

    // Paso 2: Inicializar una cola con los vertices que tienen grado de entrada cero
    queue<int> cola;
    for (int i = 0; i < nv; ++i) {
        if (gradosEntrada[i] == 0) {
            cola.push(i);
        }
    }

    vector<int> ordenTopologico; // Almacenara el orden topológico
```

Fig. 4 (Malik, D, 2013)

```
// Paso 3: Mientras la cola no este vacia
while (!cola.empty()) {
    int v = cola.front();
    cola.pop();
    ordenTopologico.push_back(v); // Agregar el vertice al orden topologico

    // Paso 4: Reducir el grado de entrada de sus vertices adyacentes
    ListaG adyacentes = g.listaAdyacencia(v);
    NodoG *actual = adyacentes.getPrimero();
    while (actual != nullptr) {
        int adyacente = actual->getDato();
        gradosEntrada[adyacente]--;
        // Si el grado de entrada del vertice adyacente llega a cero, lo agregamos a la cola
        if (gradosEntrada[adyacente] == 0) {
            cola.push(adyacente);
        }
        actual = actual->getPunt();
    }
}
```

Fig. 5 (Malik, D, 2013)

```
    // Paso 5: Verificar si el orden topologico contiene todos los vertices
    if (ordenTopologico.size() == nv) {
        // Imprimir el orden topologico como resultado del algoritmo
        cout << "Orden topologico Kahn: ";
        for (int i = 0; i < nv; ++i) {
            cout << g.getVertice(ordenTopologico[i]).getDato() << " ";
        }
        cout << endl;
    } else {
        cout << "El grafo contiene un ciclo." << endl;
    }
}
```

Fig. 6 (Malik, D, 2013)

Aplicaciones:

- Ordenamiento topológico de dependencias en proyectos.
- Resolución de problemas de planificación y programación.

Algoritmo de Búsqueda en Profundidad (DFS)

El algoritmo de búsqueda en profundidad (DFS) es una técnica utilizada para recorrer o buscar en un grafo o árbol. Puede utilizarse para múltiples propósitos, incluyendo la búsqueda de ciclos, la clasificación topológica y la generación de árboles de expansión mínima en grafos ponderados.

DFS utiliza una estrategia de exploración en profundidad, siguiendo uno de los posibles caminos hasta que alcanza la profundidad máxima antes de retroceder para explorar otros caminos. Se puede implementar de manera recursiva o utilizando una pila explícita (Fig. 7, 8).

Paso a Paso:

- Iniciar desde un vértice inicial.
- Marcar el vértice como visitado.
- Para cada vértice adyacente no visitado, repetir el proceso de DFS.
- Retornar al vértice anterior una vez que se hayan explorado todos los vértices adyacentes.
- Repetir hasta que todos los vértices sean visitados.

```
void dfs(GrafoLista &g, int v, vector<bool> &visitados, vector<int> &ordenTopologico) {
    visitados[v] = true; // Marcar el vertice como visitado

    ListaG adyacentes = g.listaAdyacencia(v); // Obtener la lista de adyacencia del vertice v
    NodoG *actual = adyacentes.getPrimero(); // Obtener el primer nodo de la lista de adyacencia

    while (actual != nullptr) {
        int adyacente = actual->getDato(); // Obtener el vertice adyacente
        if (!visitados[adyacente]) { // Si el vertice adyacente no ha sido visitado
            dfs(g, adyacente, visitados, ordenTopologico); // Llamar recursivamente a DFS
        }
        actual = actual->getPunt(); // Mover al siguiente nodo de la lista de adyacencia
    }

    // Despues de explorar todos los vertices adyacentes, agregar el vertice actual al orden topologico
    ordenTopologico.push_back(v);
}
```

Fig. 7 (Murillo, 2020)

```
void ordenamientoTopologicoDFS(GrafoLista &g) {
    int nv = g.getNumVerts(); // Obtener el numero de vertices del grafo
    vector<bool> visitados(nv, false); // Arreglo para marcar los vwrtices visitados
    vector<int> ordenTopologico; // Vector para almacenar el orden topologico

    // Recorrer todos los vertices del grafo y aplicar DFS si no han sido visitados
    for (int i = 0; i < nv; ++i) {
        if (!visitados[i]) {
            dfs(g, i, visitados, ordenTopologico); // Llamar a DFS para cada vertice no visitado
        }
    }

    // Imprimir el orden topologico obtenido
    cout << "Orden topológico DFS: ";
    for (int i = ordenTopologico.size() - 1; i >= 0; --i) {
        cout << g.getVertice(ordenTopologico[i]).getDato() << " ";
    }
    cout << endl;
}
```

Fig. 8 (Murillo, 2020)

Aplicaciones

- Determinación de conectividad entre vértices.
- Búsqueda de componentes conectados.
- Clasificación topológica en grafos dirigidos.

Estas descripciones formales proporcionan un marco claro para comprender cómo funcionan cada uno de estos algoritmos en la teoría de grafos y su aplicación en la resolución de problemas

Comparación de algoritmos y condiciones de aplicación.

En el contexto de la teoría de grafos, los algoritmos de ordenamiento topológico son fundamentales para resolver problemas relacionados con grafos dirigidos acíclicos (DAG, por sus siglas en inglés). Dos enfoques comunes para obtener el orden topológico de un DAG son el Algoritmo de Kahn y el Algoritmo de Búsqueda en Profundidad (DFS). A continuación, se presenta una comparación formal entre ambos algoritmos, así como las condiciones bajo las cuales cada uno es más apropiado:

Algoritmo de Kahn

El Algoritmo de Kahn se basa en las siguientes etapas:

- Cálculo de Grado de Entrada: Calcula el grado de entrada para cada vértice, es decir, el número de arcos que apuntan a cada vértice.
- Inicialización de la Cola: Inicializa una cola con todos los vértices que tienen un grado de entrada cero.

- Proceso de Ordenamiento:

  Mientras la cola no esté vacía:

  - Extrae un vértice de la cola.
  - Agrega este vértice al orden topológico.
  - Reduce el grado de entrada de todos sus vértices adyacentes.
  - Si algún vértice adyacente alcanza un grado de entrada cero como resultado de esta reducción, se agrega a la cola.
- Resultado: Si el orden topológico contiene todos los vértices del grafo, se imprime como resultado del algoritmo. Si no, indica que el grafo contiene un ciclo.

Condiciones de Aplicación:

- Eficiencia: Es eficiente en términos de tiempo y espacio, con una complejidad de O(V+E), donde V es el número de vértices y E es el número de arcos del grafo.
- Aplicación: Ideal para grafos donde el cálculo del grado de entrada y la manipulación de una estructura de datos tipo cola son viables y eficientes.

Algoritmo de Búsqueda en Profundidad (DFS)

El Algoritmo de Búsqueda en Profundidad se caracteriza por:

- Recursividad: Utiliza un enfoque recursivo para explorar profundamente el grafo.
- Marcaje de Visitados: Marca los vértices visitados durante la exploración.
- Ordenamiento Topológico: Después de visitar todos los vértices adyacentes de un vértice dado, lo agrega al orden topológico.

Condiciones de Aplicación

- Flexibilidad: Adecuado para grafos pequeños o medianos donde la recursión no conduce a un exceso significativo de llamadas.
- Implementación: Más sencillo de implementar que el Algoritmo de Kahn, pero puede ser menos eficiente en grafos con muchos vértices y arcos debido al uso de la recursión y la necesidad de mantener un control explícito del orden topológico.

Aplicaciones de la Ordenación Topológica

La ordenación topológica es una técnica fundamental en la teoría de grafos, utilizada para resolver diversos problemas en áreas como la planificación de tareas, compiladores, análisis de dependencias y sistemas de diseño de circuitos. A continuación, se presentan algunas aplicaciones destacadas de la ordenación topológica:

- En la gestión de proyectos, la ordenación topológica permite determinar el orden en el cual deben ejecutarse las actividades para satisfacer las dependencias entre ellas. Esto es crucial para minimizar tiempos de ejecución y optimizar recursos.
- En compiladores, la ordenación topológica se emplea para determinar el orden de compilación de módulos o archivos que dependen entre sí. Además, es útil en el análisis de dependencias entre componentes de software y la gestión de versiones.
- En redes de comunicación y sistemas de diseño de circuitos electrónicos, la ordenación topológica ayuda a comprender las interconexiones entre nodos o componentes, asegurando que todas las conexiones se establezcan correctamente sin formar ciclos.

- En problemas de optimización y resolución de restricciones, la ordenación topológica permite estructurar la evaluación de restricciones dependientes, asegurando que se cumplan todas las condiciones previas antes de aplicar una restricción particular.
- Es fundamental en algoritmos como el camino más corto en un DAG (Directed Acyclic Graph) o la detección de ciclos, donde proporciona un orden lineal de los vértices que refleja las dependencias naturales del grafo.

Casos prácticos para su uso

La Ordenación Topológica es un concepto fundamental en teoría de grafos y tiene diversas aplicaciones prácticas donde se necesite manejar dependencias y secuencias específicas. Aquí se detallan algunos casos típicos donde se puede utilizar la Ordenación Topológica:

- Gestión de Proyectos y Planificación de Tareas: En la gestión de proyectos, cada tarea puede representarse como un nodo, y las dependencias entre tareas como arcos dirigidos entre estos nodos. La ordenación topológica permite determinar el orden en el cual las tareas deben ejecutarse para cumplir con todas las dependencias, asegurando que ninguna tarea se inicie antes de que todas sus predecesoras estén completas.
- Compiladores y Análisis de Dependencias en Software: En compiladores, especialmente en la resolución de dependencias entre módulos o librerías, la ordenación topológica se usa para determinar el orden en que deben vincularse o compilarse estos módulos para garantizar que todas las dependencias estén resueltas correctamente.
- Grafos de Dependencia en Ingeniería de Software: En sistemas complejos de software, los grafos de dependencia representan las relaciones entre componentes, clases o funciones. La ordenación topológica puede ayudar a gestionar las dependencias durante la compilación, la ejecución o la actualización del software.
- Programación de Tareas en Sistemas Operativos: En sistemas operativos multitarea, la ordenación topológica se puede usar para planificar la ejecución de procesos o tareas, asegurando que se respeten las dependencias de recursos o la secuencia lógica requerida para la ejecución correcta.
- Resolución de Conflictos en Redes de Distribución y Programación de Horarios: En redes de distribución como logística o telecomunicaciones, la ordenación topológica puede optimizar la programación de rutas o la asignación de recursos, garantizando que se cumplan las restricciones de precedencia y dependencia.
- Diseño de Circuitos Electrónicos y Routing en Redes: En el diseño de circuitos electrónicos y en la configuración de routing en redes de comunicación, la ordenación topológica ayuda a establecer el flujo de datos o señales, asegurando que los nodos (como componentes electrónicos o routers) se conecten en el orden correcto.

Implementación en C++

Uso de GrafoMatriz para representación de grafos.

```
#ifndef _GRAFOMATRIZ_H
#define _GRAFOMATRIZ_H

typedef int * pint;                                                    //para el dimensionamiento de la matriz

#include "Vertice.h"                                                   //archivo de cabecera para definiciÃ³n de vÃ©rtices

class GrafoMatriz {                                                            //definiciÃ³n de la clase
private:                                                                               //definiciÃ³n de
atributos
        int maxVerts;                                                          //mÃ¡ximo numero de vÃ©rtices
        int numVerts;                                                          //nÃºmero de vÃ©rtices que contiene
el grafo
```

```
        Vertice* verts;                                         //vector de vÃ©rtices
        int** matAd;                                            //matriz de adyacencia

public:
        //mÃ©todos pÃºblicos
        GrafoMatriz();                                                  //constructor sin
parÃ¡metros
        GrafoMatriz(int v);                                             //constructor
conociendo cantidad mÃ¡xima de vÃ©rtices

                //metodos que actualizan atributos
        void setMaxVerts(int n);                                //actualiza numero maximo de vertices
        void setNumVerts(int n);                                //actualiza numero de vertices existentes en grafo
        void setVertice(int va, Vertice v);                     //modifica los atributos de un vertice, conociendo su nombre
        void setVertice(TipoG a, Vertice v);            //modifica los atributos de un vertice, conociendo su nombre
        void setArco(int va, int vb);                           //actualiza valor de arco recibiendo numeros de vertices en grafo no
valorados
        void setArco(int va, int vb, int v);            //actualiza valor de arco recibiendo numeros de vertices
        void setArco(TipoG a, TipoG b);                         //actualiza valor de arco recibiendo nombres de
vertices en grafos no valorados
        void setArco(TipoG a, TipoG b, int v);          //actualiza valor de arco recibiendo nombres de vertices

                //metodos que devuelven estado de atributos
        int getMaxVerts();                                              //devuelve numero
maximo de vertices
        int getNumVerts();                                              //devuelve cantidad de
vertices existentes en grafo
        Vertice getVertice(int va);                             //devuelve atributos de un vertice, conociendo su
numero
        Vertice getVertice(TipoG a);                            //devuelve atributos de un vertice, conociendo su
nombre
        int getArco(int va, int vb);                    //devuelve el valor de un arco recibiendo numeros de vertices
        int getArco(TipoG a, TipoG b);                          //devuelve el valor de un arco recibiendo nombres de
vertices

        int getNumVertice(TipoG v);                             //devuelve el numero de vertice conociendo su valor
        void nuevoVertice (TipoG v);                            //crea un nuevo vertice recibiendo su valor

        bool adyacente(int va, int vb);                         //determina si dos vertices son adyacentes
recibiendo numeros de vertices
        bool adyacente(TipoG a, TipoG b);                       //determina si dos vertices son adyacentes
recibiendo sus datos
};
```

Uso de GrafoLista para representación de grafos.

```
#ifndef _GRAFOLISTA_H
#define _GRAFOLISTA_H

typedef int * pint;                                     //para el dimensionamiento de la matriz

#include "Vertice.h"
#include "ListaG.h"

class GrafoLista {                                                      //definicion de la clase
private:
        //declaracion de atributos
        int maxVerts;                                                   //mÃ¡ximo numero de
vÃ©rtices
        int numVerts;                                                   //nÃºmero de
vÃ©rtices que contiene el grafo
        Vertice* verts;                                                 //vector de vÃ©rtices
        ListaG** arcos;                                                 //Lista de adyacencia

public:
        // mÃ©todos pÃºblicos de la clase GrafoMatriz
        GrafoLista();                                                   //constructor por
defecto
        GrafoLista(int v);                                              //constructor
conociendo cantidad mÃ¡xima de vÃ©rtices

        void setMaxVerts(int n);                                //actualiza numero maximo de vertices
        void setNumVerts(int n);                                //actualiza numero de vertices existentes en grafo
        void setVertice(int va, Vertice v);                     //modifica los atributos de un vertice, conociendo su nombre
        void setVertice(TipoG a, Vertice v);            //modifica los atributos de un vertice, conociendo su nombre
        void setArco(int va, int vb);                           //actualiza valor de arco recibiendo numeros de vertices en grafo
no valorados
        void setArco(int va, int vb, int v);            //actualiza valor de arco recibiendo numeros de vertices
        void setArco(TipoG a, TipoG b);                         //actualiza valor de arco recibiendo nombres de
vertices en grafos no valorados
        void setArco(TipoG a, TipoG b, int v);          //actualiza valor de arco recibiendo nombres de vertices en grafos no valorados

        int getMaxVerts();                                              //devuelve numero
maximo de vertices
        int getNumVerts();                                              //devuelve cantidad de
vertices existentes en grafo
        Vertice getVertice(int va);                             //devuelve atributos de un vertice, conociendo su
numero
        Vertice getVertice(TipoG a);                            //devuelve atributos de un vertice, conociendo su
valor
        bool getArco(int va, int vb);                   //devuelve el valor de un arco recibiendo numeros de vertices
```

```
        bool getArco(TipoG a, TipoG b);                    //devuelve el valor de un arco recibiendo valores de
vertices

        int getNumVertice(TipoG v);                        //devuelve el numero de vertice conociendo su
valor
        void nuevoVertice (TipoG v);                       //crea un nuevo vertice recibiendo su valor

        bool adyacente(int va, int vb);                    //determina si dos vertices son adyacentes
recibiendo numeros de vertices
        bool adyacente(TipoG a, TipoG b);                  //determina si dos vertices son adyacentes
recibiendo sus valores
        ListaG listaAdyacencia(int v);                     //metodo que devuelve la lista de adyacencia del
vertice v
};
```

Implementación de ordenación topológica (Matriz de adyacencia).

```
// Funcion recursiva para ingresar los vertices del grafo representado con matriz de adyacencia
GrafoMatriz ingresarVerticesMatriz(int n, int current) {
   GrafoMatriz a(n);

   if (current < n) {
      TipoG aux;
      cout << endl<<"Ingrese el valor del vertice No. " << (current + 1) << ": ";
      cin >> aux;
      a.nuevoVertice(aux);
      ingresarVerticesMatriz(n, current + 1); // llamada recursiva
   }

   return a;
}

// Funcion recursiva para ingresar los arcos del grafo representado con matriz de adyacencia
void ingresarArcosMatriz(GrafoMatriz *g, int vertice, int arco) {
   int nv = g->getNumVerts();

   if (vertice < nv) {
      Vertice x = g->getVertice(vertice);

      if (arco == 0) {
         cout <<endl<< "Cantidad de arcos de salida del vertice " << x.getDato() << ": ";
      }

      int na = leerN(0, 10);
      cin.ignore();

      if (arco < na) {
         TipoG aux;
         int peso;
         cout << "Identificador del vertice destino: ";
         cin >> aux;
         cout << "Peso del arco: ";
         peso = leerN(0, 100);
         cin.ignore();
         g->setArco(x.getDato(), aux, peso);
         ingresarArcosMatriz(g, vertice, arco + 1);
      } else {
         ingresarArcosMatriz(g, vertice + 1, 0);
      }
   }
}

// Funcion recursiva para imprimir el grafo representado con matriz de adyacencia
void imprimirGrafoMatriz(GrafoMatriz g, int vertice, int adyacente) {
   int nv = g.getNumVerts();

   if (vertice == 0 && adyacente == 0) {
      cout <<endl<< "MATRIZ DE ADYACENCIA\n";
   }

   if (vertice < nv) {
      Vertice x = g.getVertice(vertice);
      if (adyacente == 0) {
         cout <<endl<< "Vertice " << x.getDato() << ": ";
      }

      if (adyacente < nv) {
         if (g.adyacente(vertice, adyacente)) {
            Vertice y = g.getVertice(adyacente);
            cout << " -> " << y.getDato();
         }
         imprimirGrafoMatriz(g, vertice, adyacente + 1);
      } else {
         cout << endl;
         imprimirGrafoMatriz(g, vertice + 1, 0);
      }
   } else if (vertice == nv && adyacente == 0) {
      cout << "================================================\n";
   }
}
```

```
// Funcion recursiva para el ordenamiento topologico en grafo representado con matriz de adyacencia
void topoRecMatriz(GrafoMatriz &g, int v, bool *visitados, int *pila, int *tope) {
   visitados[v] = true;

   for (int i = 0; i < g.getNumVerts(); i++) {
      if (g.adyacente(v, i) && !visitados[i]) {
         topoRecMatriz(g, i, visitados, pila, tope);
      }
   }

   pila[++(*tope)] = v;
}

// Funcion para realizar el ordenamiento topologico en grafo representado con matriz de adyacencia
void ordenamientoTopologicoMatriz(GrafoMatriz &g) {
   int nv = g.getNumVerts();
   bool *visitados = new bool[nv];
   int *pila = new int[nv];
   int tope = -1;

   for (int i = 0; i < nv; i++) {
      visitados[i] = false;
   }

   for (int i = 0; i < nv; i++) {
      if (!visitados[i]) {
         topoRecMatriz(g, i, visitados, pila, &tope);
      }
   }

   while (tope >= 0) {
      cout << g.getVertice(pila[tope]).getDato() << " ";
      tope--;
   }
   cout << endl;

   delete[] visitados;
   delete[] pila;
}
```

Implementación de ordenación topológica (Lista de adyacencia)

```
// Funcion recursiva para ingresar los vertices del grafo representado con lista de adyacencia
GrafoLista ingresarVerticesLista(int n, int current) {
   GrafoLista a(n);

   if (current < n) {
      TipoG aux;
      cout << "Ingrese el valor del vertice No. " << (current + 1) << ": ";
      cin >> aux;
      a.nuevoVertice(aux);
      ingresarVerticesLista(n, current + 1);
   }

   return a;
}

// Funcion recursiva para ingresar los arcos del grafo representado con lista de adyacencia
void ingresarArcosLista(GrafoLista *g, int vertice, int arco) {
   int nv = g->getNumVerts();

   if (vertice < nv) {
      Vertice x = g->getVertice(vertice);

      if (arco == 0) {
         cout << "Cantidad de arcos de salida del vertice " << x.getDato() << ": ";
      }

      int na = leerN(0, 10);
      cin.ignore();

      if (arco < na) {
         TipoG aux;
         int peso;
         cout << "Identificador del vertice destino: ";
         cin >> aux;
         cout << "Peso del arco: ";
         peso = leerN(0, 100);
         cin.ignore();
         g->setArco(x.getDato(), aux, peso);
         ingresarArcosLista(g, vertice, arco + 1);
      } else {
         ingresarArcosLista(g, vertice + 1, 0);
      }
   }
}

// Funcion recursiva para imprimir el grafo representado con lista de adyacencia
void imprimirGrafoLista(GrafoLista g, int vertice, NodoG *adyacente) {
   int nv = g.getNumVerts();
```

```
    if (vertice == 0 && adyacente == NULL) {
        cout <<endl<< "LISTA DE ADYACENCIA\n";
    }

    if (vertice < nv) {
        Vertice x = g.getVertice(vertice);
        if (adyacente == NULL) {
            cout << "Vertice " << x.getDato() << ": ";
            adyacente = g.listaAdyacencia(vertice).getPrimero();
        }

        if (adyacente != NULL) {
            cout << " -> " << adyacente->getDato();
            imprimirGrafoLista(g, vertice, adyacente->getPunt());
        } else {
            cout << endl;
            imprimirGrafoLista(g, vertice + 1, NULL);
        }
    } else if (vertice == nv && adyacente == NULL) {
        cout << "================================================\n";
    }
}

// Funcion recursiva para el ordenamiento topologico en grafo representado con lista de adyacencia
void topoRecLista(GrafoLista &g, int v, bool *visitados, int *pila, int *tope) {
    visitados[v] = true;

    NodoG *adyacente = g.listaAdyacencia(v).getPrimero();
    while (adyacente != NULL) {
        TipoG datoAdyacente = adyacente->getDato();
        int w = g.getNumVertice(datoAdyacente);

        if (!visitados[w]) {
            topoRecLista(g, w, visitados, pila, tope);
        }
        adyacente = adyacente->getPunt();
    }

    pila[++(*tope)] = v;
}

// Funcion para realizar el ordenamiento topologico en grafo representado con lista de adyacencia
void ordenamientoTopologicoLista(GrafoLista &g) {
    int nv = g.getNumVerts();
    bool *visitados = new bool[nv];
    int *pila = new int[nv];
    int tope = -1;

    for (int i = 0; i < nv; i++) {
        visitados[i] = false;
    }

    for (int i = 0; i < nv; i++) {
        if (!visitados[i]) {
            topoRecLista(g, i, visitados, pila, &tope);
        }
    }

    while (tope >= 0) {
        cout << g.getVertice(pila[tope]).getDato() << " ";
        tope--;
    }
    cout << endl;

    delete[] visitados;
    delete[] pila;
}
```

Conclusiones

En conclusión, la ordenación topológica emerge como una técnica poderosa y versátil en la teoría de grafos, ofreciendo un método sistemático para establecer un orden lineal de los vértices en un grafo dirigido acíclico (DAG) esta ordenación no solo facilita la resolución de problemas prácticos como la planificación de tareas y la compilación de programas, sino que también asegura la coherencia y la eficiencia en la gestión de dependencias y precedencias.

La aplicación de la ordenación topológica abarca múltiples dominios, desde la ingeniería de software hasta la logística y la biología en la gestión de proyectos, por ejemplo, permite estructurar el flujo de trabajo garantizando que cada tarea se ejecute en el orden adecuado, minimizando tiempos y recursos en sistemas de software complejos, como compiladores y

sistemas de gestión de dependencias, facilita la correcta compilación y ejecución de módulos evitando errores derivados de dependencias mal resueltas.

Dos enfoques principales para obtener un orden topológico son el Algoritmo de Kahn y el Algoritmo de Búsqueda en Profundidad (DFS) cada uno con sus ventajas y aplicaciones específicas el Algoritmo de Kahn destaca por su eficiencia en términos de tiempo y espacio mientras que DFS ofrece flexibilidad y simplicidad en la implementación adaptándose bien a grafos pequeños o medianos.

Además de su utilidad en la planificación y la optimización de sistemas la ordenación topológica desempeña un papel crucial en la detección de inconsistencias estructurales la capacidad de identificar ciclos en un DAG asegura la integridad de los sistemas modelados previniendo conflictos y garantizando la consistencia de las relaciones de dependencia entre los elementos del grafo.

En conclusión, la ordenación topológica de los vértices de un grafo no solo es un concepto teórico fundamental sino también una herramienta práctica indispensable en la resolución de problemas complejos y en la optimización de procesos en una amplia gama de aplicaciones industriales y científicas su capacidad para proporcionar un orden claro y coherente de los elementos de un DAG la convierte en un componente esencial del análisis estructural y la gestión eficiente de recursos en sistemas complejos y dinámicos.

Referencias

Malik, D. S. (2013). Estructuras de datos con C++: (2 ed.). México, D.F, Mexico: Cengage Learning. Recuperado de https://elibro.net/es/ereader/espoch/39995?page=748.

3.3 ORDENAMIENTOS TOPOLÓGICOS. (2012, noviembre 18). compdiscretas. https://compdiscretas.wordpress.com/2012/11/18/3-3-ordenamientos-topologicos/

Algoritmo de Ordenamiento Topologico. (s/f). Scribd. Recuperado el 6 de julio de 2024, de https://es.scribd.com/document/226328236/Algoritmo-de-Ordenamiento-Topologico

Buchwald, A. F. C. [@algoritmos-fiuba-buchwald]. (2019, octubre 6). 07 - 06 Orden Topológico. Youtube. https://www.youtube.com/watch?v=2V6mjHFCEwc

De los grafos unidireccionales, D., Decir, E., Donde, un G., Empieces, E. D., & al nodo inicial si sigues las aristas en su direccion., no H. F. de V. (s/f). Ordenamiento topoľ ogico. Oifem.es. Recuperado el 6 de julio de 2024, de https://oifem.es/files/addendum7.pdf

Murillo, J. (2020, mayo 25). Difference between breadth search (BFS) and Deep Search (DFS). Encora. https://www.encora.com/es/blog/dfs-vs-bfs

Orden Topológico: Algoritmo de Kahn. (s/f). Github.io. Recuperado el 6 de julio de 2024, de https://arrobaricardoge.github.io/Kahn-TopoSort-Visual/

Ponderado, C. M. en un. (s/f). Tema 15 - Soluciones Greedy. Upv.es. Recuperado el 6 de julio de 2024, de https://www.grycap.upv.es/gmolto/docs/eda/EDA_Tema_15_gmolto.pdf

pythoned. (s/f). 7.17. Ordenamiento topológico — Solución de problemas con algoritmos y estructuras de datos. Runestone.academy. Recuperado el 6 de julio de 2024, de https://runestone.academy/ns/books/published/pythoned/Graphs/OrdenamientoTopologico.html