Universidad del centro de la provincia de buenos aires

Ingeñieria en sistemas

Profesores: Dra. Virginia Cifuentes, Ing. Diego Tolbaños, Dra. Virginia Yannibelli.

Alumnas: Conti Verónica, Artaza Sheila.

12 de mayo de 2023

Trabajo Práctico 1: Algoritmos sobre grafos

Análisis y Diseño de Algoritmos 2

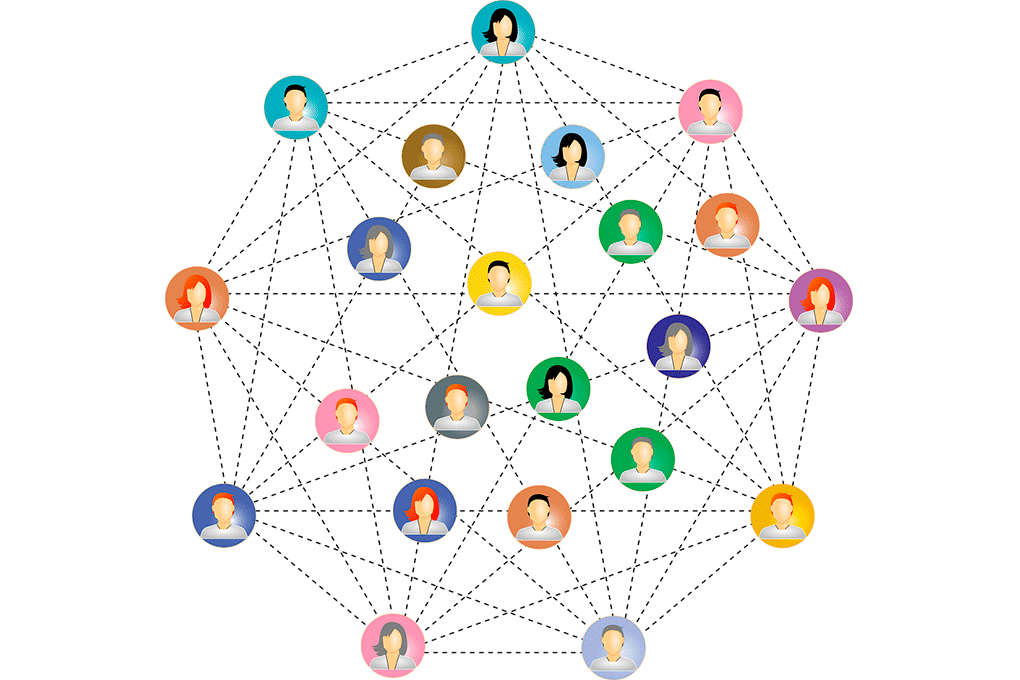
**INTRODUCCION**

En el siguiente informe detallaremos la elección, especificación, implementación, y posterior uso de las estructuras de datos Arco y Grafo para implementar un algoritmo que encuentre un camino entre dos vértices de un grafo rotulado con colores.

Para ello, las clases fueron especificadas formalmente en el lenguaje NEREUS, y luego fueron implementadas en el lenguaje C++ utilizando el IDE *Code Blocks*.

**GRAFO**

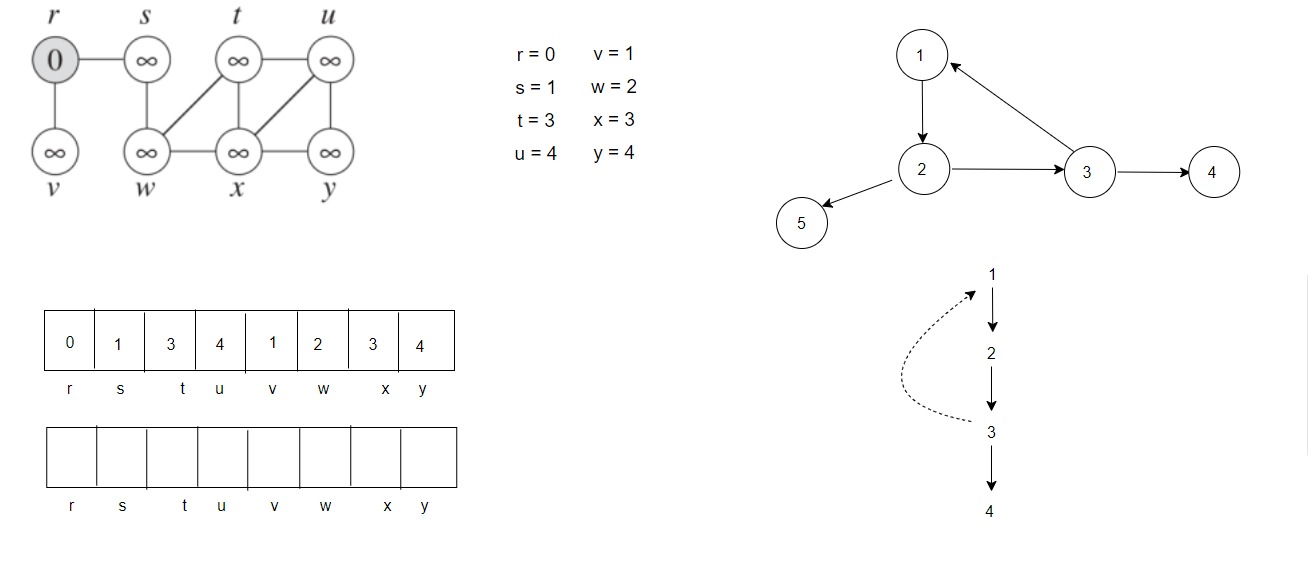
Partiendo de la definición enseñada por la cátedra, un grafo es un tipo de dato abstracto que genera una relación binaria entre objetos del mismo tipo. Son estructuras de datos dinámicas utilizadas para el análisis de redes, en el diseño de circuitos eléctricos, estrategias de mercados, cartografía, mapas conceptuales, seguridad cibernética, planificación de procesos y muchas más.



**Imagen: redes sociales mediante la teoría de grafos**

**Tipos de Grafos:**

* Dirigidos: también llamados orientados, poseen duplas (vi , vj) llamadas arcos donde un vértice vi apunta a un vértice vj . Se dice que el vértice origen es el predecesor del destino, y el destino es sucesor del origen.
* No Dirigidos: también llamados no orientados, si existe una dupla (vi , vj) también está la dupla (vj , vi) llamada arista.
* Rotulados: pueden ser tanto dirigidos como no dirigidos, pero los arcos o aristas poseen información (un rótulo) que puede ser un costo asociado a dicha conexión.
* Multígrafo: solo para los grafos no dirigidos, para un par de vértices vi y vj , hay más de una arista que los relaciona.



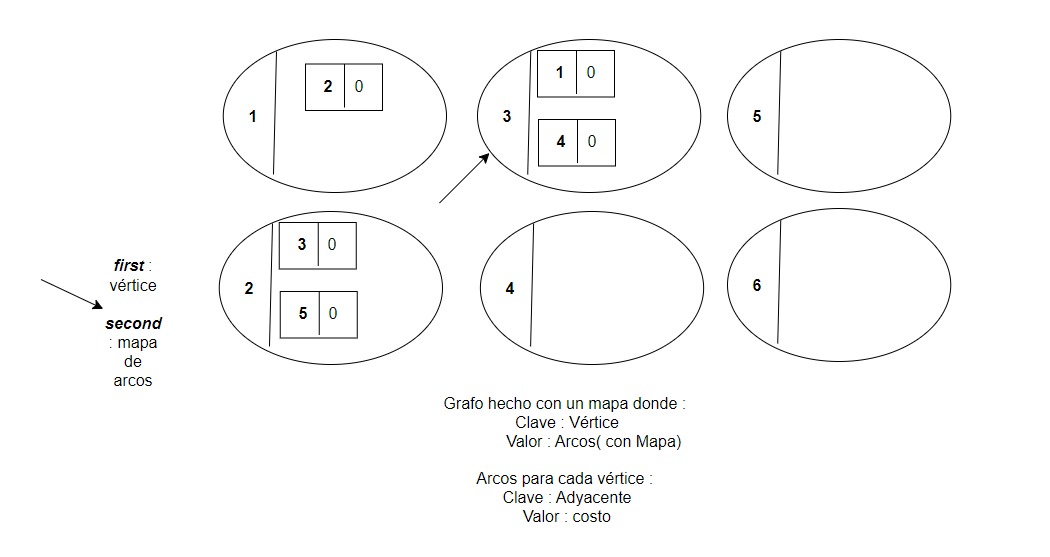
**Imagen: Conjunto de puntos (nodos o vértices) y conjuntos de líneas (flechas/arcos).**

Un camino en un grafo dirigido es una secuencia de vértices (v1 , v2 , v3 , … vk ) tal que existe un arco vi -> vi+1 para i=1,2,…,k-1. La longitud del camino es k-1 (el número de arcos).

**Implementación de Grafos**

Un grafo puede implementarse de muchas maneras. En general se presentan las primeras dos alternativas mencionadas a continuación para generarlo y una tercera opción, que es la realizada en este trabajo:

* Lista de adyacencia: dado un grafo con n vértices, se tiene un arreglo de tamaño n, donde cada celda es una lista que corresponde a los adyacentes del vértice.
* Matriz de Adyacencia: estructura estática que es de tamaño |V| x |V| y cada celda de la matriz marca con 1 si existe el arco , y 0 si no existe.
* Mapa de mapas, la estructura implementada, consiste en un contendor, mapa, que posee una primera parte: la clave, que es el vértice. Una segunda parte que es un mapa (contenedor) de arcos. A su vez los arcos en la primera sección poseen el vértice y en la segunda el valor parametrizado, en esta oportunidad es el color.

****

**Imagen: implementación del Grafo mediante mapas de mapas.**

**TDA ARCO**

CLASS ARCO;

IMPORTS;

BASIC CONSTRUCTORS Arco;

EFFECTIVE

TYPE Arco;

OPERATIONS

Arco: -> Arco;

Arco: Nat \* Nat(costo) -> Arco;

devolver\_adyacente: Arco -> nat;

devolver\_costo: Arco -> nat;

AXIOMS

//Especificación semántica.

END;

Para la implementación de la clase Arco en el ámbito privado se definieron:

* Entero que representa al adyacente. (adyacente)
* Valor parametrizado C, llamado costo que es en este caso es un entero.

**FUNCIONES DEL GRAFO Y SU COMPLEJIDAD TEMPORAL EN NOTACIÓN BIG-OH**

Arco: Constructora básica, inicializa el arco. Su complejidad temporal es O(1).

Arco: Constructora básica, agrega información del arco en cada área correspondiente. Su complejidad temporal es O(1).

devolver\_adyacente: Operación observadora, devuelve el adyacente de un arco. Su complejidad temporal es O(1).

devolver\_costo: Operación observadora, devuelve el costo del arco. Su complejidad temporal es O(1).

**TDA GRAFO**

CLASS GRAFO;

IMPORTS: Costo[Nat], lista[Arco] ;

BASIC CONSTRUCTORS Grafo;

EFFECTIVE

TYPE Grafo;

OPERATIONS

Grafo : -> Grafo;

Grafo: grafo(g) \* grafo(a) -> grafo;

esta\_vacio: grafo -> Boolean;

devolver\_longitud: grafo -> nat;

existe\_vertice: grafo \* Nat -> boolean;

existe\_arco: grafo \* Nat(origen) \* Nat(destino) -> boolean;

costo\_arco: grafo \* Nat(origen) \* Nat(destino) -> nat;

devolver\_vertices: grafo \* lista -> lista;

devolver\_adyacentes: grafo \* Nat \* lista -> lista[Arco];

agregar\_vertices: grafo \* Nat -> grafo;

eliminar\_vertice: grafo \* Nat -> grafo;

PRE: existe\_vertice(origen);

modificar\_costo\_arco: grafo \* Nat(origen) \* Nat(destino) \* Nat(costo) -> grafo;

agregar\_arco : grafo \* Nat(origen) \* Nat(destino) \* Nat(costo) -> grafo;

PRE: existe\_vertice(origen) && existe\_vertice(destino);

eliminar\_arco: grafo \* Nat(origen) \* Nat(destino) -> grafo;

PRE: existe\_arco(origen, destino);

vaciar: grafo -> grafo //PREGUNTAR

AXIOMS

//Especificación semántica

END.

Para la implementación de la clase grafo en el ámbito privado utilizamos la estructura Mapa de Mapa:

* Como clave declaramos un entero.
* Vértices: Mapa de mapa con un entero y un mapa de pares, entero y un C, siendo este el valor parametrizado que corresponde al costo.
* Construimos una función dentro del ámbito privado llamada EliminarArcosDe que es llamada por la función eliminar\_vertice y se encarga de borrar los arcos del vértice dado, si existen los mismo. Su complejidad temporal es O(a). Siendo a la cantidad de adyacentes que tenga ese vértice.

**FUNCIONES DEL GRAFO Y SU COMPLEJIDAD TEMPORAL EN NOTACIÓN BIG-OH**

Grafo: Constructora básica, inicializa el grafo. Su complejidad temporal es O(1).

Grafo: Operación constructora que asigna un grafo a otro grafo pasado por parámetro. Su complejidad temporal es O(1).

esta\_vacio: Operación observadora, retorna si esta vacío o no el grafo. Su complejidad temporal es O(1).

devolver\_longitud: Operación observadora, retorna el tamaño del grafo. Su complejidad temporal es O(1).

existe\_vertice: Operación observadora, retorna si existe cierto vértice en el grafo. Su complejidad temporal es O(n). Siendo n la cantidad de vértices.

existe\_arco: Operación observadora, retorna si existe cierto arco en el grafo. Su complejidad temporal es max(O(n,e)). Siendo n la cantidad de vértices y e el número de arcos.

costo\_arco: Operación observadora, devuelve el costo de un arco. Su complejidad temporal es max(O(n,e)). Siendo n la cantidad de vértices y e el número de arcos.

devolver\_vertices: Operación observadora, devuelve una lista con los vértices del grafo. Su complejidad temporal es O(n). Siendo n la cantidad de vértices.

devolver\_adyacentes: Operación observadora, devuelve los adyacentes de un vértice en una lista. Su complejidad temporal es max(O(n,e)). Siendo n la cantidad de vértices y e el número de arcos.

agregar\_vertice: Operación modificadora, agrega un vértice al grafo. Su complejidad temporal es O(1).

eliminar\_vertice: Operación modificadora, elimina un vértice del grafo. Su complejidad temporal es O(n.a). Siendo n la cantidad de vértices y a la cantidad de adyacentes de determinado vértice.

modificar\_costo\_arco: Operación modificadora, modifica el costo de un arco del grafo. Su complejidad temporal es max(O(n,e)). Siendo n la cantidad de vértices y e la cantidad de arcos.

agregar\_arco: Operación modificadora, agrega un arco nuevo al grafo. Su complejidad temporal es max(O(n.a)). Siendo n la cantidad de vértices y a la cantidad de adyacentes de determinado vértice. CONSULTARRRRRR COMPLEJIDAD TEMPORAL.

eliminar\_arco: Operación modificadora, elimina un arco del grafo. Su complejidad temporal es max(O(n,e)). Siendo n la cantidad de vértices y e la cantidad de arcos.

vaciar: Operación modificadora, elimina todos los vértices del grafo. Su complejidad temporal es O(n). Siendo n la cantidad de vértices.

**ALGORITMO PARA RESOLVER EL ENUNCIADO:**

Para poder resolver el trabajo decidimos utilizar una estructura auxiliar en el main que lleve el rotulo de cada vértice. El rotulo es un color y el vértice en un entero. La estructura implementada fue un mapa con un entero (vértice) y un string (color).

Con el objetivo de resolver el problema propuesto comenzamos efectuando el algoritmo recursivo DFS enseñado por la cátedra. A continuación, la primera versión del algoritmo:

DFS (Grafo g, char \*marca, int origen, int destino, list<int>& camino)

{

Marca[origen]=’G’;

Camino.agregar(origen);

If (origen== destino)

{

Imprimo(camino);

}else

List<int> adyacentes = g.adyacentes(origen);

List<int> ::const\_iterator it;

For (it=adyacentes.begin(); it!=adyacentes.end(); it++)

If (marca[\*it] ¡= ‘G’) && (\*it.devolver\_color(\*it) != ’rojo’ ))

{

DFS(g,marca,\*it,destino,camino);

}

Camino.eliminarUltimo(); //elimino de la lista de caminos asi voy buscando todos los caminos

Marca[origen]= ‘B’; //para volver a visitar vértices y otros caminos, no se si es necesario aquí ?

}

Este primer algoritmo devuelve todos los caminos desde un vértice origen hasta un destino sin pasar por vértices rotulados con el color rojo.

En esta segunda versión le agregamos el corte con un booleano haciendo que solo retorne el primer camino encontrado cuando el mismo booleano este en falso.

Segunda versión del algoritmo:

void DFS (Grafo g, char \*Marca, int origen, int destino, bool & continuar , list<int>& Camino) {

Marca[origen]='G';

Camino.push\_back(origen);

if(origen!= destino) {

List<int> adyacentes = g.devolver\_adyacentes(origen);

List<int> ::const\_iterator it = adyacentes.begin();

while ( continuar && it!=adyacentes.end() ) {

if((Marca[it->devolver\_adyacente()] != 'G') && (it->devolver\_color() != "rojo" ))

DFS(g,Marca,it->devolver\_adyacente(),destino,camino);

it++;

}

if( continuar ) {

Marca[origen] = 'B';

Camino.pop\_back();

}

}

else

continuar = false;

}

Finalmente, pasamos por parámetro el mapa con los vértices rotulados con los colores para retornar el primer camino obtenido con lo pedido en el trabajo:

void DFS (const Grafo<int> &g, char \*Marca, int origen, int destino,bool &continuar, list<int>& Camino, string color, map<int,string> coloresVert)

{

Marca[origen]='G';

Camino.push\_back(origen);

if(origen!= destino)

{

list<typename Grafo<int>::Arco> adyacentes;

g.devolver\_adyacentes(origen,adyacentes);

list<typename Grafo<int>::Arco> ::const\_iterator it= adyacentes.begin();

while (continuar && (it!=adyacentes.end())){

int vertice\_ady = it->devolver\_adyacente();

if((Marca[vertice\_ady] != 'G') && (coloresVert[vertice\_ady] != color ))

DFS(g,Marca,vertice\_ady,destino,continuar,Camino,color,coloresVert);

it++;

}

if (continuar){

Marca[origen]='B';

Camino.pop\_back();

}

}

else

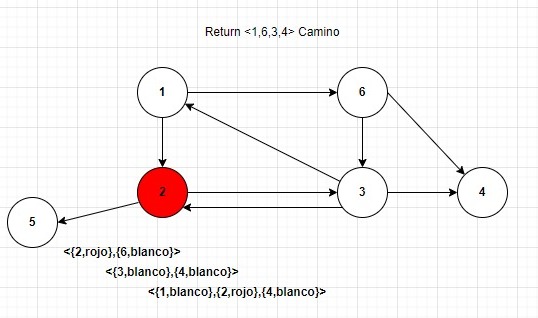
continuar=false;

}

Para efectuar el algoritmo asumimos:

* Vértices origen y destino deben ser rotulados con un color distinto a rojo.
* Grafo implementado en el código es el creado y mostrado en este informe.
* No modificar la clase grafo.
* Únicamente retornar el PRIMER camino encontrado, NO el camino más corto ni todos los caminos.

Complejidad temporal en BIG-O del algoritmo: O(n log n), siendo n la cantidad de vértices del grafo.



**Conclusión:**

Primero elaboramos la clase grafo en c++, partiendo del código propuesto en Laboratorio de Análisis y Diseño de Algoritmos. Decidimos implementarlo con mapas de mapas, con el objetivo de mejorar la complejidad temporal en los métodos del grafo. A pesar de que el código en esta estructura pueda resultar más complejo, es muy eficiente temporalmente, debido a que todas las búsquedas en esta estructura poseen una complejidad temporal logarítmica. Por otro lado, utilizando otro tipo de estructuras tales como listas de listas, matrices, entre otras, la complejidad temporal aumentaría a O(n) u O() perjudicando la eficiencia del algoritmo.

En referencia al algoritmo realizado para resolver la búsqueda de un camino partiendo desde un vértice origen hacia un destino, tal que no pase por los vértices rotulados de color rojo, fue resuelto con éxito, ya que devuelve el primer camino encontrado, pero como desventaja no siempre es el camino más corto. Al utilizar la estructura de mapa para rotular los vértices benefició a la complejidad temporal del DFS generando un acceso rápido a los rótulos.

**BIBLIOGRAFIA**

* **Paula Rochina, https://www.inesem.es/revistadigital/informatica-y-tics/teoria-grafos/;**