Laboratorio de Estructura de Datos y Algoritmos

GRAFOS

ELABORADO POR:

- *Flores Silva, David*

DOCENTE:

EDITH PAMELA RIVERO TUPAC

Ing. Informática y de Sistemas

Mgt. Seguridad Informática

Enlace del repositorio:

AREQUIPA - 2021

INFORME

En este último trabajo que está relacionado a grafos, teoría de las gráficas estudia las propiedades de los grafos (también llamadas gráficas). Un grafo es un conjunto, no vacío, de objetos llamados vértices (o nodos) y una selección de pares de vértices, llamados aristas que pueden ser orientados o no. Típicamente, un grafo se representa mediante una serie de puntos (los vértices) conectados por líneas (las aristas).

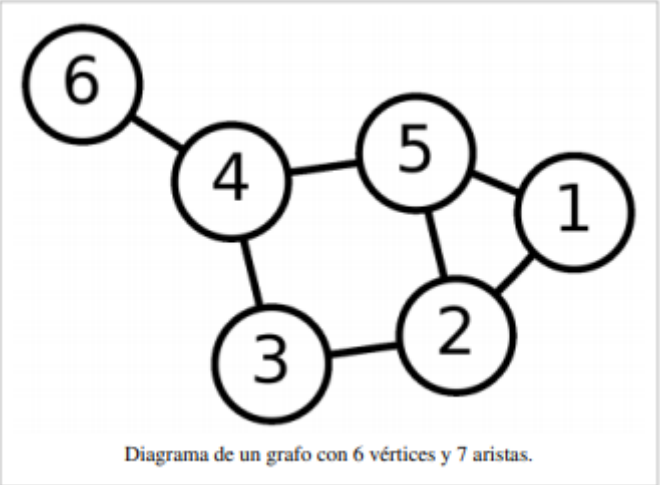

Diagrama de un grafo con 6 vértices y 7 aristas.

Estructura de lista Grafo de lista de adyacencia.

• lista de incidencia - Las aristas son representadas con un vector de pares (ordenados, si el grafo es dirigido), donde cada par representa una de las aristas.[1]

• lista de adyacencia - Cada vértice tiene una lista de vértices los cuales son adyacentes a él. Esto causa redundancia en un grafo no dirigido (ya que A existe en la lista de adyacencia de B y viceversa), pero las búsquedas son más rápidas, al costo de almacenamiento extra.

Estructuras matriciales

• Matriz de incidencia - El grafo está representado por una matriz de A (aristas) por V (vértices), donde [arista, vértice] contiene la información de la arista (1 - conectado, 0 - no conectado)

• Matriz de adyacencia - El grafo está representado por una matriz cuadrada M de tamaño , donde es el número de vértices. Si hay una arista entre un vértice x y un vértice y, entonces el elemento es 1, de lo contrario, es 0

Vértice

Los vértices constituyen uno de los dos elementos que forman un grafo. Como ocurre con el resto de las ramas de las matemáticas, a la Teoría de Grafos no le interesa saber qué son los vértices. Diferentes situaciones en las que pueden identificarse objetos y relaciones que satisfagan la definición de grafo pueden verse como grafos y así aplicar la Teoría de Grafos en ellos.

Aristas dirigidas y no dirigidas

En algunos casos es necesario asignar un sentido a las aristas, por ejemplo, si se quiere representar la red de las calles de una ciudad con sus direcciones únicas. El conjunto de aristas será ahora un subconjunto de todos los posibles pares ordenados de vértices, con $(a, b) \neq (b, a)$. Los grafos que contienen aristas dirigidas se denominan grafos orientados.

Código

en nuestro código encontraremos tres clases GRAFO, ARISTA, VÉRTICE.
sobre el código en la clase grafo tendremos nuestro la funcionalidad de agregar de unir esta clase también importa de la clase arista y de vértice.

```java
package grafos;

import grafos.Vertice;
import grafos.Arista;
import java.awt.List;
import java.util.*; //nos sire para usar la clase Vector
public class Grafos {
	public Vector lista_vert;	// es el numero de vertices que presenta el grafo
	public int numAristas = 0;	// es el numero de aristas que presenta el grafo


	//constructor inicial que llama al metodo initial()
	public Grafos() {
		init();
	}

	//el metodo init() convierte a la variable lista_vert
	//en un objeto de la clase vetor que usamos del paquete java.util.*;
	public void init() {
		lista_vert = new Vector();
	}

	/*
	  metodo que clona un grafo
	  los primero que hace es crear un objeto de la misma clase
	  luego renombra los datos de las variables
	  su numero de vertices y el numero de aristas
	  y retorna un nuevo grafo que es clonado
	*/
	public Grafos clonar() {
		Grafos g = new Grafos();
		g.lista_vert = (Vector)lista_vert.clone();
		g.numAristas = numAristas;
		return g;
	}


	//retorna la cantidad de aristas en el grafo
	public int cantVertices() {
		return lista_vert.size();
	}
```

```
63          // este metodo nos retorna los elemtosAt casteado a un objeto de tipo vertice
64          public Vertice vertice(int k) {
65                  return (Vertice)lista_vert.elementAt(k);
66          }
67
68
69
70          //el costo es lo que nos costaria ir desde una arista hasta otra arista
71          public int costo(int desde, int hasta) {
72                  Arista a = arista(desde, hasta);
73                  int r = 0;
74
75                  if( a != null ) {
76                          r = a.costo;
77                  }
78
79                  return r;
80          }
81
82
83          /*metodo que une dos aristas
84                  como primer parametros recibe un desde que es la arista de inicio
85                  como segundo parametro recibe una arista de destino
86                  como tercer paremetro recibe el costo que hay de ir desde
87                  el primer parametro hasta el segundo parametro
88          */
89          public int unirAristas(int desde, int hasta, int costo) {
90                  Arista newArista = new Arista(desde, hasta, costo);
91                  unirAristas(desde, hasta, newArista);
92                  return ++numAristas;
93          }
94
95
96          /*metodo que une dos aristas
97                  como primer parametros recibe un desde que es la de inicio
98                  como segundo parametro recibe una arista de destino
99                  como tercer paremetro recibe un valor de tipo Arista que es la arista que se tendra que conectar
100          */
```

```
101         public void unirAristas(int desde, int hasta, Arista a) {
102                 if( desde >= cantVertices() ) {
103                         lista_vert.setSize(desde+1);
104                 }
105
106                 if( hasta >= cantVertices() ) {
107                         lista_vert.setSize(hasta+1);
108                 }
109
110                 if( vertice(desde) == null ) {
111                         lista_vert.set(desde, new Vertice(null));
112                 }
113
114                 if( vertice(hasta) == null ) {
115                         lista_vert.set(hasta, new Vertice(null));
116                 }
117
118                 vertice(desde).unirAristas(a);
119         }
120
121
122         public void separarAristas(int desde, int hasta) {
123                 if(cantVertices() > desde) {
124                         vertice(desde).separarAristas(hasta);
125                 }
126         }
127
128
129
130         public int agregarVertice(String id) {
131                 Vertice newVertice = new Vertice(id);
132                 lista_vert.add(newVertice);
133
134                 return cantVertices()-1;
135         }
136
137         public void agregarVertice(String id, int pos) {
138                 if(pos >= cantVertices() ) {
139                         lista_vert.setSize(pos+1);
140                 }
141
142                 Vertice newVertice = new Vertice(id);
143                 lista_vert.set(pos, newVertice);
144         }
```

```java
package grafos;

public class Arista {
	public int costo=0;
	public int desde;
	public int hasta;

	public Arista(int desde, int hasta, int costo) {
		this.desde = desde;
		this.hasta = hasta;
		this.costo = costo;
	}

	public String toString() {
		return "costo = "+costo;
	}


}
```

```java
package grafos;

import java.util.*;

public class Vertice {
        public String id;
        public ArrayList<Arista>  lista_ady;
        public boolean visitado;


        public Vertice(String id) {
                nombrar(id);
                lista_ady= new ArrayList<>();
                visitado = false;
        }

        public void unirAristas ( Arista a ) {
        ListIterator Litr = lista_ady.listIterator(0);
        int pos = 0;

        while (Litr.hasNext()) {
            Arista ari = (Arista) Litr.next();

            if (a.hasta < ari.hasta) {
                break;
            }
            pos++;
        }

        lista_ady.add(pos,a);
    }

        public void separarAristas ( int to) {
            ListIterator Litr = lista_ady.listIterator(0);

```

```
33          public void separarAristas ( int to) {
34              ListIterator Litr = lista_ady.listIterator(0);
35
36          while (Litr.hasNext()) {
37              Arista a = (Arista) Litr.next();
38
39              if (a.hasta == to) {
40                  Litr.remove();
41                  break;
42              }
43          }
44      }
45
46          public Arista adyacente(int to) {
47          ListIterator AdyItr =  lista_ady.listIterator(0);
48          Arista a=null;
49
50          while ( AdyItr.hasNext() ) {
51              a = (Arista) AdyItr.next();
52              if (a.hasta == to){
53                  return a;
54              }
55          }
56
57              return null;
58      }
59
60      public void nombrar (String id) {
61          this.id = id;
62          }
63  }
```