Apunte 5 - Grafos

Índice general

Índice de figuras

Algoritmos

Índice de ejercicios

Apunte 5

Grafos

Actualizado: 23 de junio de 2020

En esta unidad se trabajará sobre la siguiente pregunta:

> *¿Cuáles son las estructuras de datos adecuadas para modelar relaciones arbitrarias entre datos, cómo se pueden implementar y en qué situaciones de la vida real son aplicables?*

A lo largo de este capítulo se estudiará la estructura de datos Grafo; de la cual se presentarán los conceptos básicos, usos y operaciones más importantes. También se presentarán distintas implementaciones, analizando en cada caso el uso de memoria y el tiempo de ejecución de las distintas operaciones.

5.1. Definición de grafos

Un grafo es un conjunto de objetos llamados *vértices* o *nodos* unidos por enlaces llamados *aristas* o *arcos*, que permiten representar relaciones arbitrarias entre elementos de un conjunto. Visualmente, un grafo se representa como un conjunto de puntos (los vértices) unidos por líneas (aristas).

En la práctica los grafos permiten estudiar interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de computadoras puede representarse y estudiarse mediante un grafo, en el cual los vértices representan terminales y las aristas representan conexiones (las cuales, a su vez, pueden ser cables o conexiones inalámbricas).

Como modelo, un grafo G es un par ordenado G=(V,E), donde:

- V es un conjunto de vértices o nodos,
- E es un conjunto de aristas o arcos, que relacionan estos nodos.

Existen dos tipos principales de grafo:

- *Grafo no dirigido* o *grafo propiamente dicho*: es un grafo G = (V, E) donde:
  - V es un conjunto no vacío,
  - E es un conjunto de pares no ordenados de elementos de V. Un par no ordenado es un conjunto de la forma {a, b}, de manera que $a, b = b, a$.
- *Grafo dirigido o dígrafo* es un grafo G = (V, E) donde:
  - V es un conjunto no vacío
  - E es un conjunto de pares ordenados de elementos de V, de manera que $a, b \neq b, a$
  - Dada una arista (a,b), a es su nodo inicial y b su nodo final.

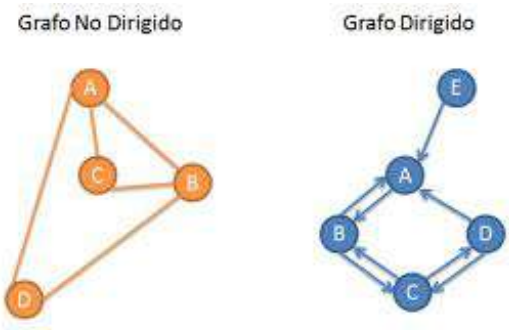


Figura 5.1: Ejemplos de grafo y dígrafo no etiquetados

La estructura *grafo* puede permitir o no múltiples aristas entre cada par de vértices. Si se quiere remarcar la inexistencia de múltiples aristas entre cada par de vértices el grafo puede llamarse *simple*. Por otra parte, si se quiere asegurar la posibilidad de permitir múltiples aristas, se llamarán *multigrafo* (si no es dirigido) o *multidigrafo* (si es dirigido). En la Figura 5.2 se muestran ejemplos de ambos tipos de grafo.

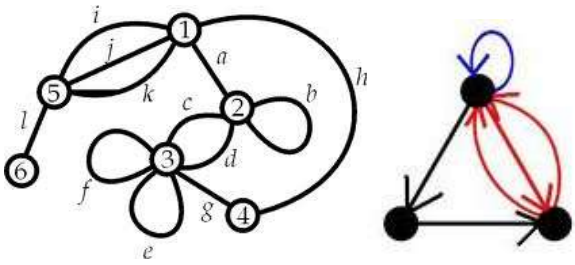
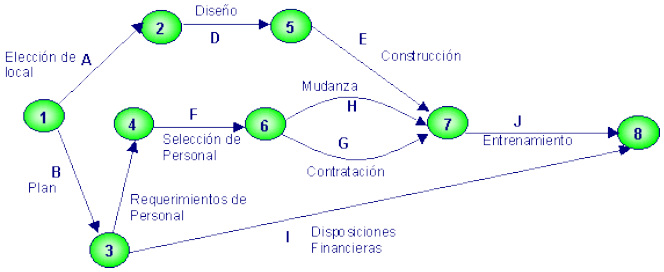


Figura 5.2: Ejemplos de multigrafos

Otra manera de clasificar los grafos es si son etiquetados o no. Un grafo *etiquetado* es aquel que permite guardar información en las aristas. En la Figura 5.3 se muestran ejemplos de un grafo y un dígrafo etiquetados. En el grafo de la izquierda (A), los vértices representan ciudades y la etiqueta se utiliza para indicar la distancia en kilómetros entre ellas. En el dígrafo de la derecha (B), los vértices representan etapas de un proyecto y las etiquetas representan las actividades necesarias para alcanzar cada etapa del mismo.

(A) Red vial



(B) Red de actividades

Figura 5.3: Ejemplos de grafos etiquetados

A continuación se presentan otras definiciones importantes de grafo, utilizando como ejemplo los grafos de la Figura 5.3.

- Se llama *orden* del grafo G a su número de vértices, $|V|$. Ejemplos: el orden de (A) es 17 y el orden de (B) es 8.
- El *grado de un vértice* $v \in V$ es igual al número de arcos que lo tienen como extremo. Ejemplos: En (A), el grado del vértice Madrid es 6. En (B) el grado del vértice 7 es 4. Además, en un dígrafo, el

grado puede diferenciarse en *grado de salida* (cantidad de arcos que tienen al vértice como origen) y *grado de entrada* (cantidad de arcos que lo tienen como destino). En el ejemplo (B), el vértice 7 tiene grado de entrada 3 y grado de salida 1.

- En un grafo, dos vértices son *adyacentes* si los une una arista. Ejemplo: en (A), Granada y Sevilla son adyacentes porque hay una ruta que las une.
- En un dígrafo, el vértice destino del arco es *adyacente* del vértice origen. Ejemplo: La tarea 8 es adyacente a 7 (pero no a la inversa, porque no hay arco desde 8 a 7).
- Una arista es *incidente* a un vértice si ésta lo une a otro. Ejemplo: en (B) el arco "Plan" es incidente en las tareas 1 y 3.
- Un *camino* es una sucesión de arcos que permiten llegar desde un vértice inicial a un vértice final. En el ejemplo de la red vial (Figura 5.3), se puede encontrar más de un camino para llegar desde Valladolid a Murcia. El camino puede enunciarse por los arcos o los vértices que lo forman, según la información que se desee recuperar del grafo. Por ejemplo, un camino formado por los vértices (Valladolid, Madrid, Albacete, Murcia), sirve para saber por qué ciudades se debe pasar para llegar de un punto a otro. En cambio, si se desea calcular la distancia, será más adecuado recuperar la información de los arcos (193, 251, 150). En el ejemplo se ve que puede haber más de un camino posible entre dos vértices, pues también puede llegarse de Valladolid a Murcia visitando (Valladolid, Bilbao, Zaragoza, Barcelona, Valencia, Murcia). Según el tipo de modelo que represente el grafo puede interesar el camino de menor longitud en kilómetros (suma de los pesos de los arcos) o el de menos escalas (vértices).
- Un *ciclo* es un camino que permite volver al vértice de origen. Por ejemplo, en la Figura 5.3, el camino (Madrid, Albacete, Murcia, Granada, Jaén, Madrid), es un ciclo ya que comienza y termina en Madrid.

5.2. Recorridos en grafos

A continuación se presentarán dos formas de recorrer un grafo que permiten pasar por todos los vértices del mismo, pero visitando cada vértice una sola vez.

5.2.1. Búsqueda en profundidad

Una búsqueda en profundidad (en inglés DFS o Depth First Search) es un algoritmo que permite recorrer todos los vértices de un grafo de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir visitando todos los vértices que va encontrando en un camino hacia "adelante". Cuando ya no le quedan más vértices por visitar en dicho camino, regresa (haciendo backtracking) y se fija si le quedó algún vértice adyacente sin visitar y reanuda la búsqueda hacia adelante. Repite el proceso hasta que haya visitado todos los nodos.

Este recorrido es análogo al recorrido en preorden de un árbol genérico, y como tal, es recursivo. La clave de este recorrido es que se lleva una estructura auxiliar donde se guardan los vértices que ya fueron visitados, para no volver a pasar por ellos.

Algoritmo 5.1 Algoritmo de recorrido en profundidad de grafo

```
Algoritmo Profundidad (G)
    crear estructura visitados
    para cada vértice u del grafo G hacer
        si u no está en visitados entonces
            ProfundidadDesde(u, visitados)
        fin si
    fin para
Fin algoritmo
Modulo ProfundidadDesde(u, visitados)
    insertar u en visitados
    para cada vértice v adyacente de u hacer
        si v no está en visitados entonces
            ProfundidadDesde(v, visitados)
        fin si
    fin para
Fin modulo
```

En la Figura 5.4 se muestra un ejemplo de recorrido en profundidad del dígrafo del ejemplo.

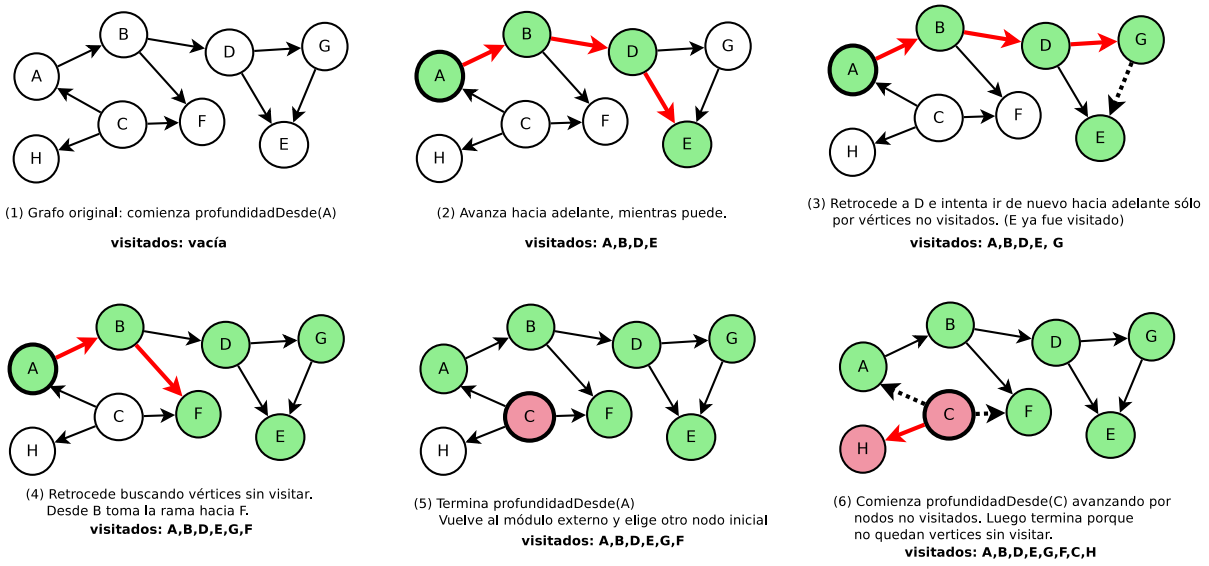


Figura 5.4: Ejemplo de ejecución del algoritmo de búsqueda en profundidad

En la primera ejecución del algoritmo se toma el vértice A, y se invoca al módulo recursivo *ProfundidadDesde*(A). En el módulo recursivo se marca a A como visitado y se toma uno de sus adyacentes (en este caso sólo B) y se repite el mismo algoritmo, visitando B,D y E. Como E no tiene adyacentes, se termina la ejecución de ese módulo y se vuelve al piso inferior de la pila recursiva, donde quedó pendiente la ejecución del vértice D. Como D tiene un vértice adyacente aún sin visitar, el algoritmo avanza hacia G. Desde G, como tiene adyacente a E pero éste ya fue visitado, no puede continuar, entonces termina la ejecución. Volviendo hacia atrás en la recursión, descubre que B tiene el adyacente F sin visitar, entonces avanza hacia él. Como F no tiene adyacentes, termina la ejecución. A continuación termina la ejecución del módulo B y luego la del vértice A inicial. Al terminar la ejecución de *ProfundidadDesde*(A), retorna al algoritmo principal y busca si quedó algún vértice sin visitar. Puede tomar C o H. Suponiendo que tomara a C, al invocar *ProfundidadDesde*(C) visitaría a C y H y terminaría porque no quedan más vértices sin visitar.

Es importante dejar en claro que la acción de "elegir" un vértice para continuar el recorrido (sea el vértice inicial o un adyacente) en la implementación no es aleatorio sino que depende del orden en que los vértices se almacenen en la estructura del grafo.

5.2.2. Búsqueda en anchura

La búsqueda en anchura (en inglés *BFS - Breadth First Search*) es un algoritmo para recorrer o buscar elementos en un grafo, que (como el recorrido en profundidad) visita cada vértice una sola vez. Se comienza eligiendo un vértice arbitrario y se exploran primero todos sus vecinos (adyacentes). A continuación, para cada uno de sus vértices adyacentes se exploran sus respectivos adyacentes, y así hasta cubrir todos los vértices alcanzables desde el vértice inicial. Este recorrido es análogo al recorrido por niveles de árbol genérico, y como tal, utiliza una cola para saber por cuál vértice debe continuar. Además utiliza una lista para saber cuáles son todos los vértices visitados hasta el momento y evitar entrar en ciclos.

Algoritmo 5.2 Algoritmo de recorrido en anchura de grafo

```
Algoritmo RecAnchura (G)
    crear estructura visitados
    para cada vértice u del grafo G hacer
        si u no está en visitados entonces
            AnchuraDesde(u, visitados)
        fin si
    fin para
Fin algoritmo
Algoritmo AnchuraDesde (verticeInicial, visitados)
    crear cola Q
    insertar v en visitados
    poner verticeInicial en Q
    mientras Q no esté vacía hacer
        u = obtener el frente de Q
        sacar el elemento del frente de Q
        para cada vértice v adyacente de u hacer
            si v no está en visitados entonces
                insertar v en visitados
                poner v en Q
            fin si
        fin para
    fin mientras
Fin algoritmo
```

En la Figura 5.5 se muestra un ejemplo de recorrido en anchura del dígrafo del ejemplo. Al ejecutar el algoritmo *RecAnchura*, se toma el vértice B para comenzar y se invoca al módulo *AnchuraDesde*(B). Internamente se saca a B de la cola auxiliar, se marca a B como visitado y se pone cada uno de sus adyacentes (F y D) en la cola. Continúa sacando el frente de la cola (F) que como no tiene adyacentes sólo se lo marca como visitado. Luego saca a D de la cola, se lo marca como visitado y se insertan sus adyacentes (E y G) en la cola. Se saca G de la cola, se lo marca como visitado y como no tiene adyacentes, se continúa sacando a E que tampoco tiene adyacentes. Al quedar la cola vacía termina la ejecución de *AnchuraDesde*(B). Al volver al modulo *RecAnchura*, se elige otro vértice sin visitar (C) y se ejecuta *AnchuraDesde*(C) de manera similar. Se sigue hasta que no queden vértices del grafo sin visitar.

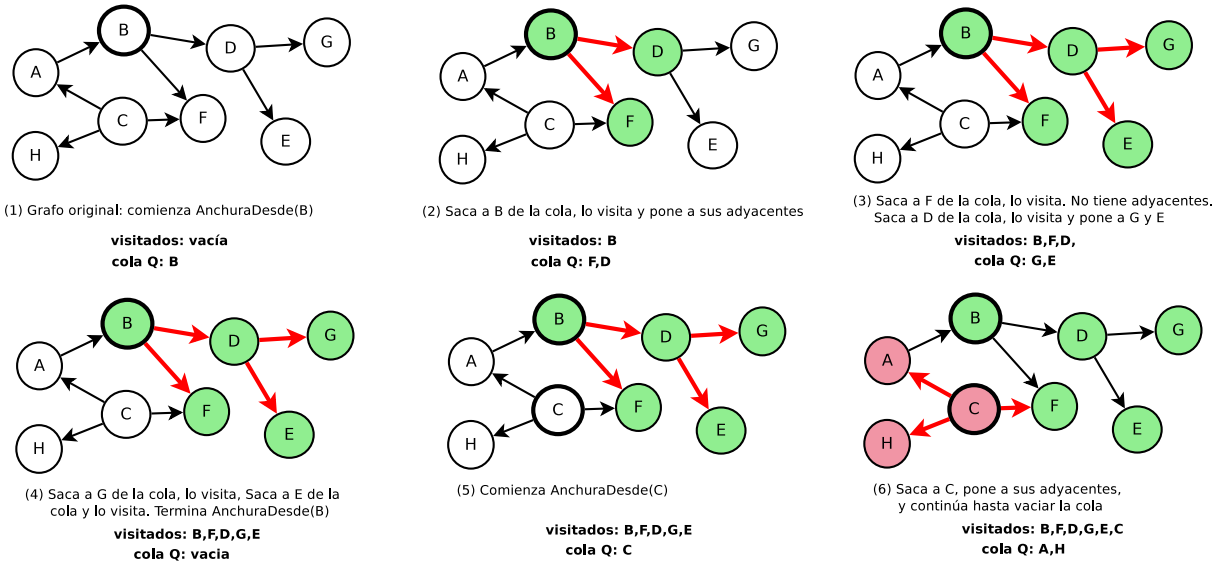


Figura 5.5: Ejemplo de ejecución del algoritmo de búsqueda en anchura

5.3. Operaciones del TDA Grafo

Sea *TipoVertice* el tipo de elemento almacenado en los vértices del grafo:

- *constructor vacío*:
  // Crea un grafo vacío
- *insertarVertice*: TipoVertice → boolean
  // Dado un elemento de TipoVertice se lo agrega a la estructura controlando que no se inserten vértices repetidos. Si puede realizar la inserción devuelve *verdadero*, en caso contrario devuelve *falso*.
- *eliminarVertice*: TipoVertice → boolean
  // Dado un elemento de TipoVertice se lo quita de la estructura. Si se encuentra el vértice, también deben eliminarse todos los arcos que lo tengan como origen o destino. Si se puede realizar la eliminación con éxito devuelve *verdadero*, en caso contrario devuelve *falso*.
- *insertarArco*: origen, destino → boolean [1]
  // Dados dos elementos de TipoVertice (origen y destino) agrega el arco en la estructura, sólo si ambos vértices ya existen en el grafo. Si puede realizar la inserción devuelve *verdadero*, en caso contrario devuelve *falso*.
- *eliminarArco*: origen, destino → boolean [2]
  // Dados dos elementos de TipoVertice (origen y destino) se quita de la estructura el arco que une ambos vértices. Si el arco existe y se puede realizar la eliminación con éxito devuelve *verdadero*, en caso contrario devuelve *falso*.
- *existeVertice*: TipoVertice → boolean
  // Dado un elemento, devuelve *verdadero* si está en la estructura y *falso* en caso contrario.
- *existeArco*: origen, destino → boolean
  // Dados dos elementos de TipoVertice (origen y destino), devuelve *verdadero* si existe un arco en la estructura que los une y *falso* en caso contrario.
- *existeCamino*: origen, destino → boolean
  // Dados dos elementos de TipoVertice (origen y destino), devuelve *verdadero* si existe al menos un camino que permite llegar del vértice origen al vértice destino y *falso* en caso contrario.
- *caminoMasCorto*: origen, destino → Lista (de elementos TipoVertice)
  // Dados dos elementos de TipoVertice (origen y destino), devuelve un camino (lista de vértices) que indique el camino que pasa por menos vértices que permite llegar del vértice origen al vértice destino. Si hay más de un camino con igual cantidad de vértices, devuelve cualquiera de ellos. Si alguno de los vértices no existe o no hay camino posible entre ellos devuelve la lista vacía.

[1] Si se trata de un grafo etiquetado, se agregará un tercer parámetro con la etiqueta
[2] Si se trata de un multigrafo etiquetado, se deberá agregar un parámetro indicando la etiqueta del arco a eliminar

- *caminoMasLargo*: origen, destino → Lista (de elementos TipoVertice)
  // Dados dos elementos de TipoVertice (origen y destino), devuelve un camino (lista de vértices) que indique el camino que pasa por más vértices (sin ciclos) que permite llegar del vértice origen al vértice destino. Si hay más de un camino con igual cantidad de vértices, devuelve cualquiera de ellos. Si alguno de los vértices no existe o no hay camino posible entre ellos devuelve la lista vacía.

- *listarEnProfundidad*: → Lista (de elementos TipoVertice)
  // Devuelve una lista con los vértices del grafo visitados según el recorrido en profundidad explicado en la sección anterior.

- *listarEnAnchura*: → Lista (de elementos TipoVertice)
  // Devuelve una lista con los vértices del grafo visitados según el recorrido en anchura explicado en la sección anterior.

- *esVacio*: → boolean
  // Devuelve *falso* si hay al menos un vértice cargado en el grafo y *verdadero* en caso contrario.

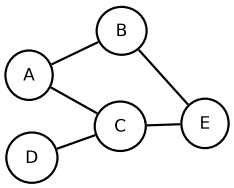
- *clone*: → Grafo
  // Genera y devuelve un grafo que es equivalente (igual estructura y contenido de los nodos) al original.

- *toString*: → String
  // Con fines de debugging, este método genera y devuelve una cadena String que muestra los vértices almacenados en el grafo y qué adyacentes tiene cada uno de ellos.

5.4. Implementaciones

5.4.1. Matriz de adyacencia

La matriz de adyacencia es una manera sencilla de representar grafos. Dada un matriz M de tamaño NxN de tipo boolean, se puede representar un grafo de N vértices. Si es un grafo dirigido, la celda M[i,j] indica si existe un arco entre el vértice i y el vértice j. Si existe un arco (i,j), la celda está en *true* y *false* en caso contrario. Si se usa para modelar un grafo no dirigido, para una arista (i,j) se inicializan a *true* ambas celdas M[i,j] y M[j,i], por lo que la matriz será *simétrica*. En la Figura 5.6 se muestra un ejemplo de un grafo no etiquetado representado con matriz de adyacencia.



	A	B	C	D	E
A	false	true	true	false	false
B	true	false	false	false	true
C	true	false	false	true	true
D	false	false	true	true	false
E	false	true	false	false	false

Figura 5.6: Representación de grafo no etiquetado con matriz de adyacencia

Para representar un multigrafo o multidigrafo, se suele utilizar una matriz de tipo entero (en lugar de boolean) y en la celda se almacena la cantidad de arcos, como se muestra en la Figura 5.7.

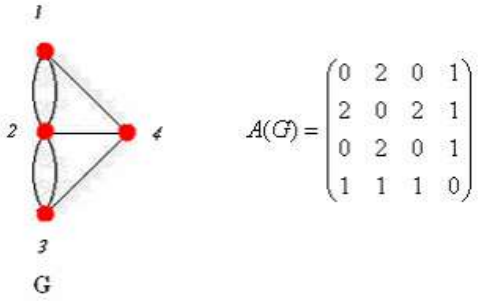
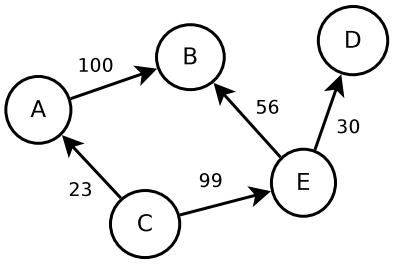


Figura 5.7: Representación de multigrafo no etiquetado con matriz de adyacencia

También es posible representar un grafo o dígrafo etiquetado utilizando la misma representación, pero cambiando el tipo de elemento de la matriz por el tipo de la etiqueta. Por ejemplo, en la Figura 5.8 se observa un dígrafo etiquetado, donde el tipo de la matriz es entero. Al ser un dígrafo, los arcos son almacenados solamente en la fila del vértice origen del arco.



	A	B	C	D	E
A	0	100	0	0	0
B	0	0	0	0	0
C	23	0	0	0	99
D	0	0	0	0	0
E	0	56	0	30	0

Figura 5.8: Representación de dígrafo etiquetado con matriz de adyacencia

En la Figura 5.9 se muestra un diagrama de clases UML para la implementación de un grafo no etiquetado que permite almacenar vértices de *tipoElemento*.

La definición de la clase presentada en 5.9 es común para grafo o dígrafo: los métodos que deberán revisarse y ajustarse de manera adecuada son *insertarArco* y *eliminarArco*, recordando que en el dígrafo un arco se almacena en una sola dirección y en un grafo en ambas direcciones. En esta implementación, la operación de eliminar un vértice debe analizarse en detalle, ya que no es un caso trivial.

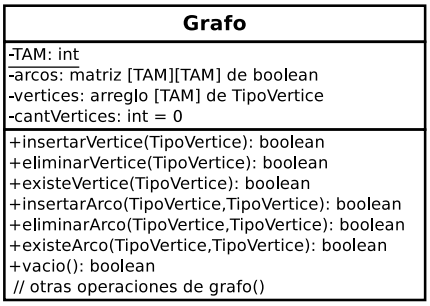
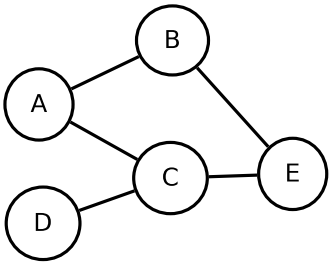


Figura 5.9: Diagrama de clases UML de Grafo (con matriz de adyacencia)

Por ejemplo, la implementación anterior podría utilizarse para almacenar el grafo de la Figura 5.8, donde los nombres de los vértices (A,B,C,D,E) se almacenan en el arreglo de manera paralela a la matriz, esto significa que si el vértice B está almacenado en la posición 1 del arreglo *vertices*, en la fila 1 de la matriz *arcos* se almacenarán sus adyacentes. Los arcos se almacenan en la matriz, como valores booleanos en las intersecciones de los vértices adyacentes. En la Figura 5.10 se muestra el grafo de dicho ejemplo almacenado de acuerdo a esta implementación en Java, donde el arreglo sólo puede tomar índices enteros mayores o iguales a cero.



	0	1	2	3	4
0	false	true	true	false	false
1	true	false	false	false	true
2	true	false	false	true	true
3	false	false	true	true	false
4	false	true	false	false	false

arcos

0	A
1	B
2	C
3	D
4	E

vertices

Figura 5.10: Representación de grafo con tipo vértice genérico con matriz de adyacencia en Java

Si se desea implementar grafos etiquetados, la matriz se definirá de *tipoEtiqueta* en lugar de tipo boolean, y se redefinirá el método *insertarArco* para que tenga el parámetro *etiqueta*.

La representación de matriz de adyacencia tiene la ventaja del acceso de O(1) a cualquier arco del grafo. Por lo general, es la representación adecuada cuando los vértices son del tipo de los índices del arreglo, pero resulta poco práctica cuando se necesita almacenar vértices de otro tipo. Como suele ocurrir con las implementaciones estáticas, tiene la desventaja de ocupar más espacio de memoria del que generalmente se necesita. Para solucionar los problemas de la implementación estática, a continuación se presentará una implementación más flexible.

5.4.2. Listas de adyacencia

La representación de grafos con listas de adyacencia es una implementación dinámica. Se utilizan dos tipos de nodos:

- *Nodo Vértice*: se crea un nodo de este tipo para cada vértice del grafo. Se enlazan formando una lista, comenzando por un nodo cabecera llamado *inicio* que es almacenado en la clase Grafo. Cada nodo de tipo Nodo Vértice almacena el nombre que identifica al vértice, un enlace al siguiente vértice y un enlace al primer vértice adyacente, que se representará con un Nodo Adyacente.
- *Nodo Adyacente*: se crea un nodo de este tipo para cada arco del grafo. Estos nodos se enlazan al vértice origen del arco. Este nodo almacena un enlace al Nodo Vértice destino del arco y un enlace al próximo nodo adyacente. Si se trata de un grafo/dígrafo etiquetado, la etiqueta se agrega en este nodo.

5.4.2.1. Implementación con listas de adyacencia de grafo no etiquetado

En la Figura 5.11 se observa el diagrama de clases para esta implementación. Como se explicó para la implementación de matriz de adyacencia, la definición de la clase es igual para grafo o dígrafo, debiéndose ajustar la implementación de los métodos *insertarArco* y *eliminarArco*.

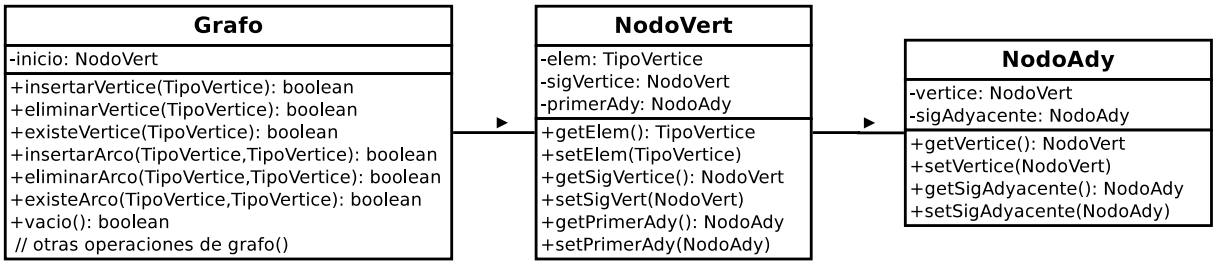


Figura 5.11: Diagrama de clases UML para Grafo no etiquetado (implementación dinámica)

En la Figura 5.12 se muestra un ejemplo de un dígrafo implementado de manera dinámica, donde el *tipoElemento* (de *NodoVert*) es char. En la figura de la izquierda se observa el dígrafo que se quiere representar, y en la figura de la derecha se observa cómo se forma la estructura de grafo. La estructura tiene 4 nodos de tipo *NodoVert* (uno para cada vértice del grafo) que están enlazados entre ellos formando una lista. Para reconocer mejor los tipos de los nodos, los de tipo *NodoVert* se han pintado de amarillo y los de tipo *NodoAdy* son un poco más pequeños y se han pintado de celeste. Por ejemplo, para representar los adyacentes del vértice A, el nodo está enlazado con 2 nodos de tipo *NodoAdy*, el primero referencia al *NodoVert* de B y el siguiente apunta al *NodoVert* de C. Estos enlaces (que se han pintado de rojo) permiten acceder más rápido al adyacente que si se guardara su nombre y hubiera que buscarlo en la lista de vértices.

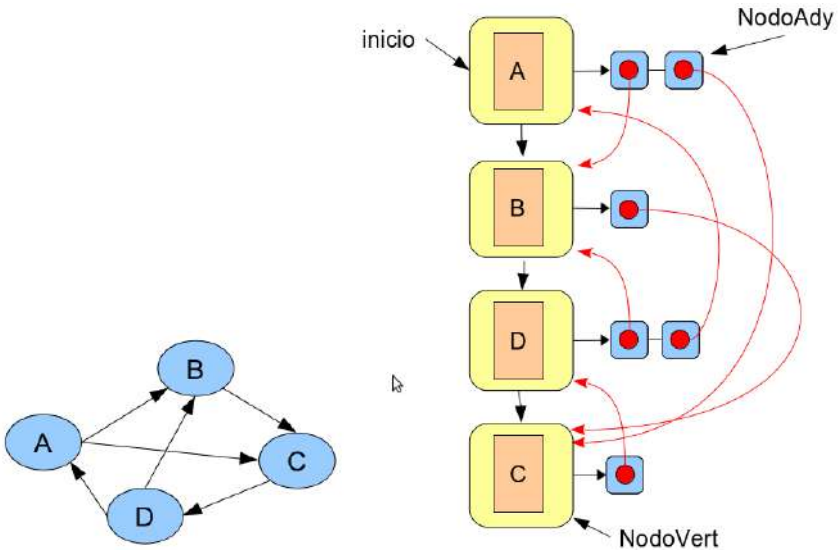


Figura 5.12: Representación de dígrafo no etiquetado con listas de adyacencia

A continuación se presenta la definición de clases Java necesarias para implementar un grafo no etiquetado, con vértices de tipo Object.

Algoritmo 5.3 Definición de clases para Grafo no etiquetado con vértices tipo Object (con listas de adyacencia)

```
public class GrafoNoEtiq {

    // atributo
    private NodoVert inicio = null;
```

```
class NodoVert {

    // atributos
    private Object elem;
    private NodoVert sigVertice;
    private NodoAdy primerAdy;
```

```
class NodoAdy {

    // atributos
    private NodoVert vertice;
    private NodoAdy sigAdyacente;
```

A continuación se presenta el método que inserta un nuevo vértice en un grafo definido en la clase *GrafoNoEtiq* presentada anteriormente. El método *insertarVertice* verifica que el vértice a insertar no esté repetido. Si no está lo agrega al principio de la lista de vértices. Se utiliza un método auxiliar privado *ubicarVertice* que recorre la lista de vértices buscando un elemento, y si lo encuentra devuelve el enlace al nodo que lo contiene. Este método será útil para otras operaciones, por ejemplo, para verificar que los vértices origen y destino de un arco existen antes de insertar o eliminar dicho arco.

Algoritmo 5.4 Código Java del método *insertarVertice* en Grafo (con listas de adyacencia)

```
private NodoVert ubicarVertice(Object buscado) {
    NodoVert aux = this.inicio;
    while (aux != null && !aux.getElem().equals(buscado)) {
        aux = aux.getSigVertice();
    }
    return aux;
}

public boolean insertarVertice(Object nuevoVertice) {
    boolean exito = false;
    NodoVert aux = this.ubicarVertice(nuevoVertice);
    if (aux == null) {
        this.inicio = new NodoVert(nuevoVertice, this.inicio);
        exito = true;
    }
    return exito;
}
```

Respecto al recorrido recursivo de un grafo, a continuación se presenta el código del método *listarEnProfundidad*. El método público comienza por el vértice inicial, invocando a un método recursivo privado (*listarEnProfundidadAux*) que hace el recorrido en profundidad desde dicho vértice. Una vez agotado el recorrido en profundidad a partir de un vértice, el método público busca si queda algún vértice que aún no haya sido visitado y vuelve a comenzar desde allí. Ambos métodos comparten el uso de una lista de elementos (visitados), que en este caso es también la lista que el método público devuelve.

Algoritmo 5.5 Código Java del método *listarEnProfundidad* (con listas de adyacencia)

```
public Lista listarEnProfundidad() {
    Lista visitados = new Lista();
    // define un vertice donde comenzar a recorrer
    NodoVert aux = this.inicio;
    while (aux != null) {
        if (visitados.localizar(aux.getElem()) < 0) {
            // si el vertice no fue visitado aun, avanza en profundidad
            listarEnProfundidadAux(aux, visitados);
        }
        aux = aux.getSigVertice();
    }
    return visitados;
}

private void listarEnProfundidadAux(NodoVert n, Lista vis) {
    if (n != null) {
        // marca al vertice n como visitado
        vis.insertar(n.getElem(), vis.longitud() + 1);
        NodoAdy ady = n.getPrimerAdy();
        while (ady != null) {
            // visita en profundidad los adyacentes de n aun no visitados
            if (vis.localizar(ady.getVertice().getElem()) < 0) {
                listarEnProfundidadAux(ady.getVertice(), vis);
            }
            ady = ady.getSigAdyacente();
        }
    }
}
```

En el siguiente ejemplo se observa el uso del recorrido en profundidad para verificar si existe un camino entre dos vértices. En este caso, el método público verifica que ambos vértices son válidos y a continuación trabaja de manera recursiva, utilizando la lista de visitados para evitar recorrer vértices más de una vez. En caso que en el recorrido en profundidad se llegue al vértice destino deseado, el método deja de invocarse recursivamente y devuelve el valor verdadero a su llamador.

Algoritmo 5.6 Código Java del método *existeCamino* (con listas de adyacencia)

```
public boolean existeCamino(Object origen, Object destino) {
    boolean exito = false;
    // verifica si ambos vertices existen
    NodoVert auxO = null;
    NodoVert auxD = null;
    NodoVert aux = this.inicio;

    while ((auxO == null || auxD == null) && aux != null){
        if (aux.getElem().equals(origen)) auxO=aux;
        if (aux.getElem().equals(destino)) auxD=aux;
        aux = aux.getSigVertice();
    }

    if (auxO != null && auxD != null) {
        // si ambos vertices existen busca si existe camino entre ambos
        Lista visitados = new Lista();
        exito = existeCaminoAux(auxO, destino, visitados);
    }
    return exito;
}
private boolean existeCaminoAux(NodoVert n, Object dest, Lista vis) {
    boolean exito = false;
    if (n != null) {
        // si vertice n es el destino: HAY CAMINO!
        if (n.getElem().equals(dest)) {
            exito = true;
        } else {
            // si no es el destino verifica si hay camino entre n y destino
            vis.insertar(n.getElem(), vis.longitud() + 1);
            NodoAdy ady = n.getPrimerAdy();
            while (!exito && ady != null) {
                if (vis.localizar(ady.getVertice().getElem()) < 0) {
                    exito = existeCaminoAux(ady.getVertice(), dest, vis);
                }
                ady = ady.getSigAdyacente();
            }
        }
    }
    return exito;
}
```

5.4.2.2. Implementación de grafos etiquetados con listas de adyacencia

En la Figura 5.13 se observa el diagrama de clases para un grafo etiquetado. Como se explicó antes, la etiqueta se agrega en el nodo adyacente (*NodoAdy*).

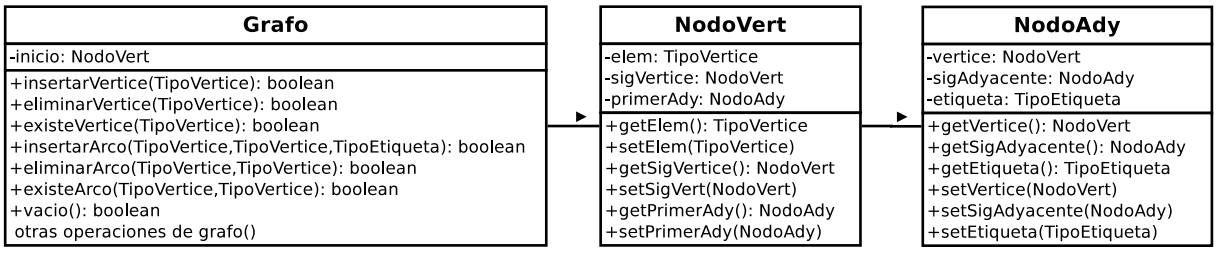


Figura 5.13: Diagrama de clases UML para Grafo etiquetado (implementación dinámica)

En la Figura 5.14 se muestra un ejemplo de un dígrafo etiquetado implementado de manera dinámica, donde *TipoElemento* (de *NodoVert*) es char y *tipoEtiqueta* (en *NodoAdy*) es int. En la figura de la izquierda se observa el dígrafo que se quiere representar, y en la figura de la derecha se observa cómo se forma la estructura de grafo con los nodos enlazados. Como se explicó anteriormente, la estructura tiene un nodo de tipo *NodoVert* para cada vértice del grafo, enlazados entre ellos formando una lista. En este caso, también se han pintado de amarillo los nodos de tipo *NodoVert* y de celeste los de tipo *NodoAdy*. Las etiquetas de los arcos se han agregado en los nodos de tipo *NodoAdy*, de acuerdo al UML presentado en la Figura 5.13.

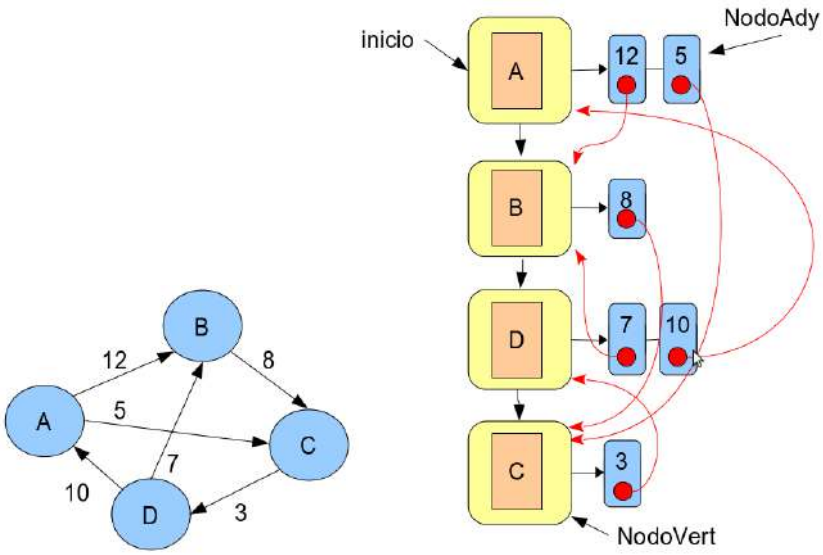


Figura 5.14: Representación de dígrafo etiquetado con listas de adyacencia

---

Ejercicio 5.1: Implementación dinámica de Grafo no dirigido

- Crear una clase que implemente el TDA Grafo no etiquetado para elementos de tipo genérico.
- Realizar una clase de testing para la clase anterior.

5.5. Análisis de eficiencia

La ventaja de la representación estática es que es muy sencilla si los vértices son de un tipo que pueda ser índice de la matriz. La eficiencia será mayor si el grafo tiene muchos arcos entre sus vértices. A medida que las celdas de la matriz tengan más valores *false* (es decir que tiene pocos arcos entre los vértices), la matriz ocupará más espacio del necesario. Las matrices que tienen 30 % o menos de sus celdas en valores distintos de vacío, se llaman ralas y siempre se recomienda utilizar una implementación alternativa.

Por otro lado, las ventajas de la representación dinámica son las siguientes:

- La implementación es similar para vértices de cualquier tipo (a diferencia de la implementación estática que, al menos en Java, sólo sería eficiente para vértices de tipo entero $>= 0$).
- La cantidad de vértices puede crecer y decrecer durante la ejecución, simplemente agregando un nuevo nodo a la lista de vértices
- La lista de adyacentes de cada vértice también puede crecer o decrecer como sea necesario, aunque en este caso a mayor cantidad de adyacentes menor eficiencia para recorrer el grafo.

5.6. Ejemplos de aplicaciones de grafos

Los grafos son estructuras adecuadas para modelar una amplia variedad de casos reales. A continuación se presentan algunos ejemplos típicos:

- *Redes de transporte*: los vértices representan lugares (estaciones de subterráneo, estaciones de trenes, aeropuertos, ciudades, direcciones en una ciudad); y los arcos representan las rutas que los unen. En estos grafos el objetivo principal suele ser encontrar distintos tipos de caminos entre dos vértices: el camino más corto, todos los caminos posibles, los caminos que pasan por cierto vértice, etc. Según el dominio, puede usarse un grafo o un dígrafo. Por ejemplo, para modelar las líneas de subte (ejemplo 1), como los trenes hacen el recorrido de ida y de vuelta, el modelo básico es un grafo. En un mapa callejero (ejemplo 2) se debe considerar la dirección de las calles para poder ir en coche (dígrafo),

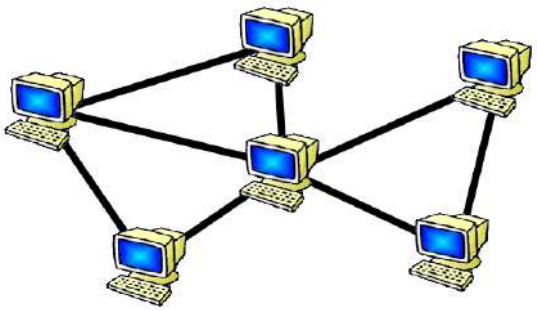
o no si se desea ir caminando (grafo). El ejemplo 3 muestra un sitio web que vende pasajes aéreos y que busca combinaciones de vuelos entre dos ciudades o aeropuertos. Dado que cada vuelo tiene aeropuerto origen y destino, se modelaría usando un dígrafo.

Ejemplo 1: Red de subterráneos

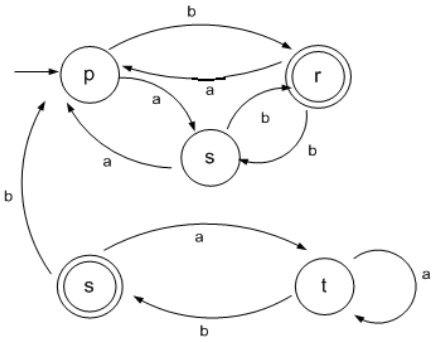
Ejemplo 2: Mapa callejero

Ejemplo 3: Venta de pasajes aéreos

- *Usos en ciencias de la computación y sistemas*:
  - *Modelado de redes de computadoras*: Los vértices son las computadoras y los arcos representan el cableado o la posibilidad de conexión entre dos computadoras (ejemplo 4). Almacenando la información en un grafo es posible identificar, por ejemplo, si una computadora es vital para mantener a todas las computadoras conectadas, es decir si al dejar de funcionar ocasiona que la red quede partida en 2 o más grupos de computadoras desconectados entre sí (algoritmo de detección de puntos de articulación).
  - *Máquinas de estados finitos*: Un autómata finito es un modelo matemático de una máquina que acepta cadenas de un lenguaje definido sobre un alfabeto predeterminado. Los vértices del dígrafo representan el conjunto finito de estados y los arcos las transiciones posibles entre dichos estados (ejemplo 5).



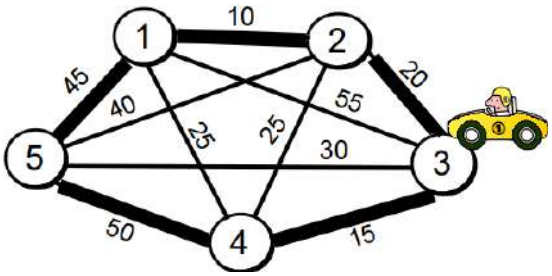
Ejemplo 4: Red de computadoras



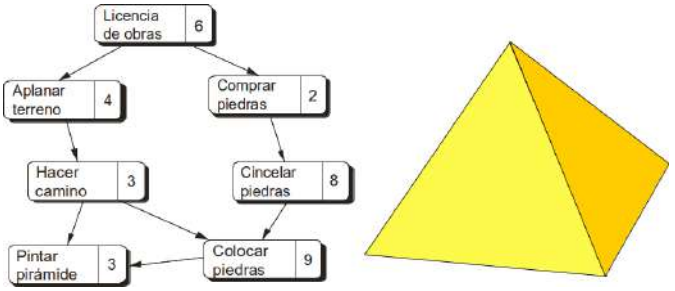
Ejemplo 5: Máquinas de estados finitos

- *Algoritmos de optimización*: Los grafos también sirven como base para resolver muchos problemas de optimización.
  - *El problema del viajante*: Los vértices representan las ciudades por donde un viajante debe pasar y los arcos representan la distancia entre ellas (ejemplo 6). El problema que se intenta resolver es encontrar el recorrido más corto en distancia que visite todas las ciudades, pasando una sola vez por cada una y volviendo al punto de origen. Este problema fue formulado por primera vez en 1930 y es uno de los problemas de optimización más estudiados.
  - *Redes de actividades*: Este tipo de representación tiene una variada gama de aplicaciones dentro de la administración moderna y la industria de la construcción. Las redes de actividades son dígrafos acíclicos cuyos vértices representan actividades y los arcos representan una relación de precedencia entre dos actividades (ejemplo 7). Si hay un arco desde la actividad A a la actividad B, significa que la actividad A debe terminar antes de comenzar a ejecutar la actividad B. En la figura del ejemplo 7, la actividad "Comprar piedras" debe terminar antes de comenzar la

actividad "Cincelar piedras". En este tipo de grafo, uno de los problemas fundamentales suele ser encontrar el *camino crítico*, que es la secuencia de actividades que ocupan el mayor tiempo de ejecución del proyecto y suelen definir la duración total del mismo. A partir de cálculos sobre la red de actividades se obtiene un tiempo probabilístico de terminación del proyecto.

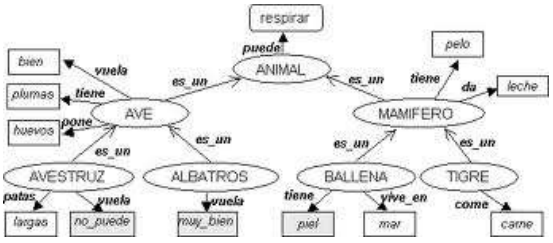


Ejemplo 6: Problema del viajante

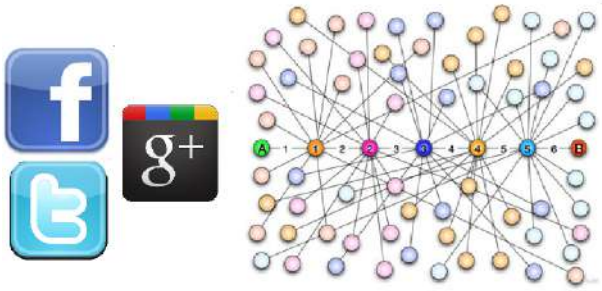


Ejemplo 7: Red de actividades

- *Usos en otras ciencias*:
  - *Mapa conceptual*: es una técnica usada para la representación gráfica del conocimiento. Un mapa conceptual es una red donde los vértices representan los conceptos y los arcos representan las relaciones entre ellos. Se usan principalmente en actividades pedagógicas.
  - *Análisis de redes sociales*: Una red social representa una estructura social por medio de un grafo. El análisis de las redes sociales estudia la estructura social aplicando la teoría de grafos e identificando las entidades (personas, usuarios) como vértices y las relaciones entre ellos como aristas (parentesco, amistad, relación laboral, etc). El análisis de redes sociales ha emergido como una metodología clave en las ciencias sociales modernas (sociología, antropología, psicología social, economía y ciencias políticas, entre otras).



Ejemplo 8: Mapa conceptual



Ejemplo 9: Análisis de redes sociales

5.7. Bibliografía sobre grafos

- A.V. Aho, J.E. Hopcroft y J.D. Ullman, Estructuras de Datos y Algoritmos, Addison Wesley Iberoamericana, 1988.