Estructuras de Datos no Lineales. Grafos

Versión 1.0

CAMAVINGAGII

2025

ÍNDICE GENERAL

CAPÍTULO

CONCEPTOS BÁSICOS DE GRAFO

1

Definición 1.0.1

Un **grafo** es un conjunto en el que hay definido una relación binaria, es decir, $G = (V, A)$, tal que V son el conjunto de **vértices o nodos** y A son un conjunto de **aristas o arcos** $A \subseteq (VxV)$ que define una relación binaria en V, es decir, que une dos vértices. Cada arista es por tanto, un par de vértices $(v, w) \in A$

- Básicamente son un conjunto de vértices relacionados/conectados entre sí por líneas llamadas aristas.
- La existencia/inexistencia de arista entre dos vértices implica la existencia/inexistencia de relación entre los mismos.

1.1 Grafo dirigido

- Si las líneas que interconectan los vértices tienen un sentido diremos que el grafo es **dirigido**. Del vértice que sale la flecha es el vértice **incidente** y al que le apunta es **adyacente**. A estas líneas que poseen una dirección se le denominan **arcos**

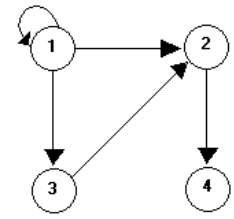

Figura 1.1: Ejemplo grafo dirigido

1.2 Grafo no dirigido

- Si por el contrario cada línea no posee una dirección, entenderemos que es bidireccional, es decir, se puede ir de A a B y de B a A. En ese caso, diremos que el grafo es **no dirigido**. Los vértices que une la línea son **adyacentes** y a la línea se le denomina **arista**. Estas aristas no poseen dirección, por lo que si vemos un grafo cuyas líneas no poseen punta de flecha, son aristas, y se entiende que son bidireccionales.

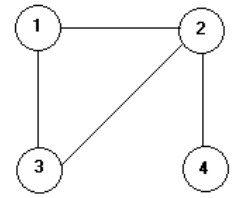

Figura 1.2: Ejemplo grafo no dirigido

1.3 Pesos de aristas. Grafo ponderado

- Una arista puede tener un valor asociado, denominado **peso**, que representa un tiempo, una distancia, un coste... Un grafo cuyas aristas poseen valores recibe el nombre de **grafo ponderado**

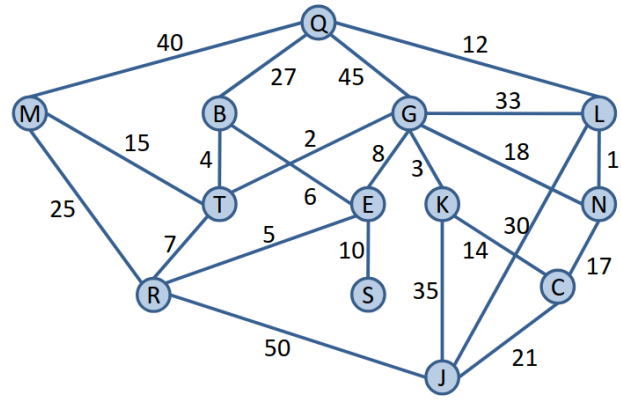

Figura 1.3: Ejemplo aristas con peso

CAPÍTULO 2

DEFINICIONES VARIAS

2.1 Grado

- El **grado** de un vértice en un grafo no dirigido es el número de aristas del vértice. Si el grafo es dirigido, se distingue entre **grado de entrada** (número de arcos que apuntan hacia el vértice) y **grado de salida** (número de arcos adyacentes al vértice).

2.2 Camino

- Es una sucesión de vértices de un grafo, n_1, n_2, n_k tal que (n_i, n_i+1) es una arista para 1<i<k. La **longitud** del camino es el número de arcos que comprende en este caso k-1. (es obvio, si tengo 3 nodos, para llegar de A a C, pasando obligatoriamente por B, me hacen falta 2 aristas o arcos). Si el grafo es ponderado la longitud de un camino se calcula como la suma de los pesos de las aristas que lo constituyen.

2.2.1. Camino simple

- Un camino es simple cuando **sus arcos son todos distintos**. Si además todos los vértices son distintos, se llama **camino elemental**

2.3 Ciclo

- Es un camino en el que coinciden los vértices inicial y final. Si el camino es simple, el ciclo es **simple** y si el camino es elemental, entonces el ciclo es **elemental**. Se permiten arcos de un vértice a sí mismo, cuya longitud será 0. Si un grafo contiene arcos de la forma (v,v), tendrá longitud 1. (o del peso asociado al arco)

2.4 Grafo conexo

- Un grafo es **conexo** cuando existe al menos, un camino entre cualquier par de vértices.

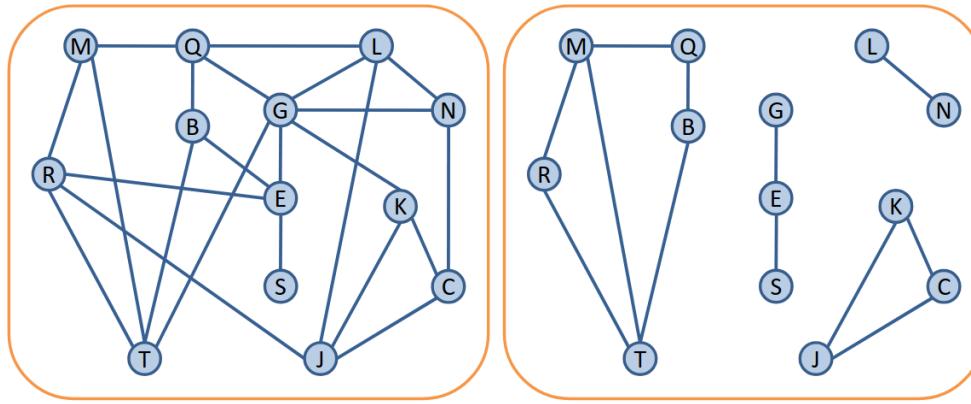

Figura 2.1: Ejemplos grafos conexo y no conexo

2.4.1. Grafo fuertemente conexo

- Es un grafo dirigido en el que hay al menos un camino entre cualquier par de vértices. Si un grafo no es fuertemente conexo, pero si tomamos el grafo como un grafo no dirigido, y es conexo, entonces decimos que el grafo es **débilmente conexo**.

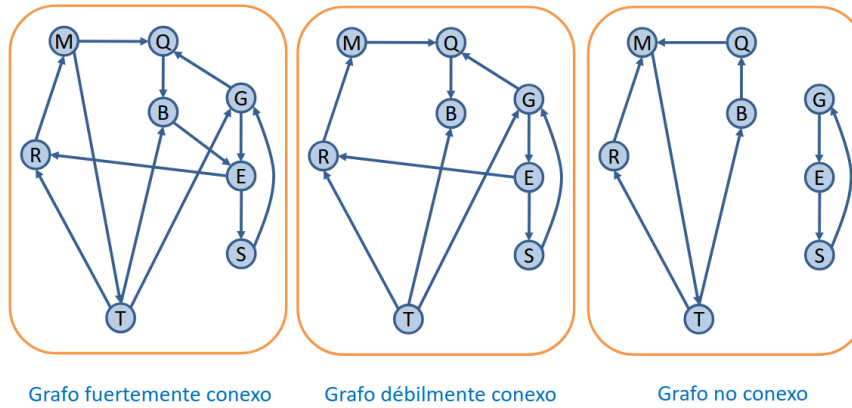

Figura 2.2: Ejemplos varios

2.5 Grafo completo

- Aquel en el cual existe una arista entre cualquier par de vértices (en ambos sentidos si el grafo es dirigido). Es decir, cada vértice tiene aristas a todos los demás vértices.
- El número de aristas será para grafos dirigidos n y para no dirigidos: $\frac{n(n-1)}{2}$

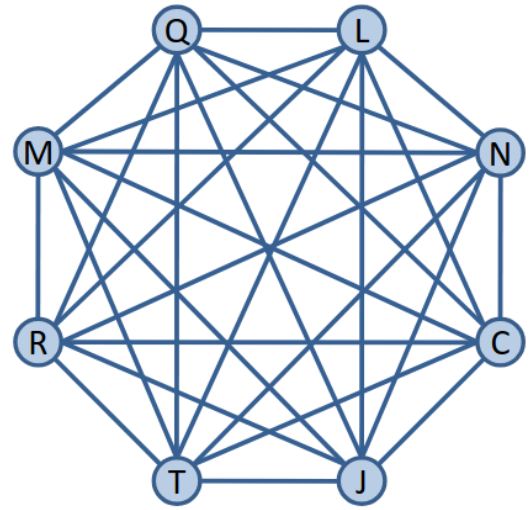

Figura 2.3: Ejemplo grafo completo

2.6 Subgrafo

- Dado un grafo G= (V,A) diremos que G' = (V',A'), donde $V' \subseteq V$ y $A' \subseteq A$, es un subgrafo de G si A' solo contiene **todas las aristas de A que unen vértices de V'**. Es decir, si cogemos unos vértices de G, esto será un subgrafo del mismo siempre y cuando mantengamos las aristas que los unen.

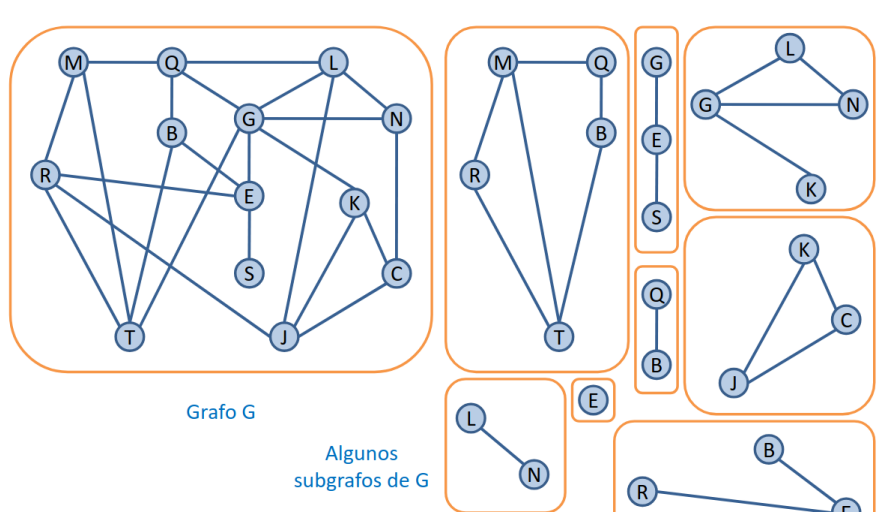

Figura 2.4: Ejemplos subgrafos

2.7 Componente conexo

- Un **componente conexo** es un subgrafo donde todos los vértices está conectados entre sí y no hay forma de llegar a nodos que están fuera de ese subgrafo.
  Es decir, es un grupo de nodos que forman un grafo conexo por sí solo y que no están conectados con otros subgrafos.

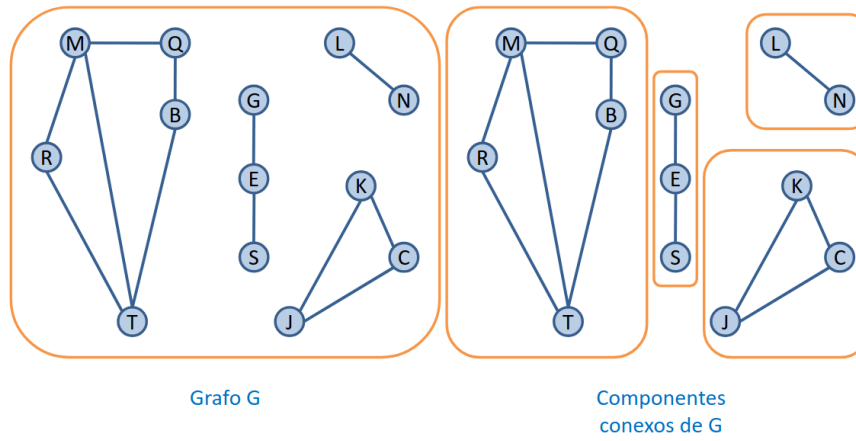

Figura 2.5: Ejemplo componente conexo

CAPÍTULO

3 REPRESENTACIONES DE GRAFOS

3.1 Matriz de adyacencia

- Dado un grafo G=(V,A) con n vértices, se define la matriz de adyacencia asociada a G como una Matriz nxn donde M(i,j)=1 si $(i,j) \in A$ y M(i,j)=0 si $(i,j) \notin A$. Es decir, en la posición i,j habrá un 1 (true) si existe una arista entre el vértice i y el vértice j
- Si G es un grafo no dirigido, la matriz es simétrica, ya que (i,j)=(j,i) para cualquier par de vértices i,j. (por tanto bastaría con recorrer la mitad de la matriz)

```
#include <vector>

class Grafo {

    public:
    typedef size_t vertice; // un valor entre 0 y Grafo::numVert()-1
    explicit Grafo(size_t n): ady(n, vector<bool>(n, false)) {}
    size_t numVert() const {return ady.size();}
    const vector<bool>& operator [](vertice v) const {return ady[v];}
    vector<bool>& operator [](vertice v) {return ady[v];}

    private:
    vector< vector<bool> > ady;
};
```

3.2 Matriz de costes

- Dado un grafo G=(V,A) con n vértices, se define la matriz de costes asociada a G como una matriz C de nxn donde C(i,j)=p si para el par de vértices i,j existe un peso asociado, y C(i,j)=peso_nulo(o ilegal) si no existe un peso asociado para el par de vértices i,j.

```
#include <vector>
#include <limits>

template <typename T> class GrafoP { // Grafo ponderado

    public:
    typedef T tCoste;
    typedef size_t vertice; // un valor entre 0 y GrafoP::numVert()-1
    static const tCoste INFINITO; // peso arista inexistente
    explicit GrafoP(size_t n): costes(n, vector<tCoste>(n,INFINITO)){}
    size_t numVert() const {return costes.size();}
    const vector<tCoste>& operator [](vertice v) const {return costes[v
        ];}
    vector<tCoste>& operator [](vertice v) {return costes[v];}
    bool esDirigido() const;

    private:
    vector< vector<tCoste> > costes;

};

    // Definicion de INFINITO o peso nulo
    template <typename T>
    const tCoste GrafoP<T>::INFINITO = std::numeric_limits<T>::max();
```

3.3 Listas de adyacencia

- Asociamos a cada vértice i del grafo una lista que almacena todos los vertices adyacentes a i. En los grafos ponderados se guarda para cada vértice una lista con los vértices adyacentes y su peso.

```
#include <vector>
#include "listaenla.h"

class Grafo {
    //grafos NO ponderados
    public:
    typedef size_t vertice; // un valor entre 0 y Grafo::numVert()-1
    explicit Grafo(size_t n): ady(n) {}
    size_t numVert() const {return ady.size();}
    const Lista<vertice>& adyacentes(vertice v) const {return ady[v];}
    Lista<vertice>& adyacentes(vertice v) {return ady[v];}

    private:
    vector< Lista<vertice> > ady; // vector de listas de vertices
    };
```

```
    //grafos ponderados

#include <vector>
#include "listaenla.h"

template <typename T> class GrafoP { // Grafo ponderado

    public:
    typedef T tCoste
    typedef size_t vertice; // un valor entre 0 y GrafoP::numVert()-1
    struct vertice_coste { // vertice adyacente y coste
    vertice v;
    tCoste c;
    // requerido por Lista<vertice_coste>::buscar()
    bool operator ==(const vertice_coste& vc) const {return v == vc.v;}
    };
    static const tCoste INFINITO; // peso de arista inexistente
    GrafoP(size_t n): ady(n) {}
    size_t numVert() const {return ady.size();}
    const Lista<vertice_coste>& adyacentes(vertice v) const {return ady[v
        ];}
    Lista<vertice_coste>& adyacentes(vertice v) {return ady[v];}

    private:
    vector<Lista<vertice_coste> > ady; // vector de listas de vertice-
        coste


};
//definicion de infinito
```

```
29 template <typename T>
30 const tCoste GrafoP<T>::INFINITO = std::numeric_limits<T>::max();
```

CAPÍTULO 4

REPRESENTACIONES DE GRAFOS

4.1 Ventajas e inconvenientes

- Las matrices de adyacencia y costes son muy eficientes para comprobar si existe una arista entre un vértice y otro. (ya que es acceder a una matriz por indexación)
- Pueden desaprovechar gran cantidad de memoria si el grafo no es completo. (si tengo una matriz de 1000x1000 con un millón de nodos posibles, si tengo solo dos nodos en el grafo desaprovecho espacio)
- La representación mediante listas de adyacencia aprovecha mejor el espacio de memoria, pues sólo se representan los arcos existentes en el grafo.
- Las listas de adyacencia son poco eficientes para determinar si existe una arista entre dos vértices del grafo. (ya que para cada vértice tengo que recorrer la lista para ver si existe o no una arista con otro vértice).
- Todas estas estructuras de datos no admiten la adición y eliminación de vértices, si no se utilizan matrices y vectores dinámicos.
- Una estructura alternativa es una lista de listas de adyacencia, en la que es posible añadir y suprimir vértices. (es decir, en vez de un vector, una lista, donde para cada elemento de una lista es otra lista donde se almacenan los vértices adyacentes al vértice correspondiente)