Estructuras de Datos no Lineales

Tema 2. Grafos

José Fidel Argudo Argudo
José Antonio Alonso de la Huerta
Mª Teresa García Horcajadas

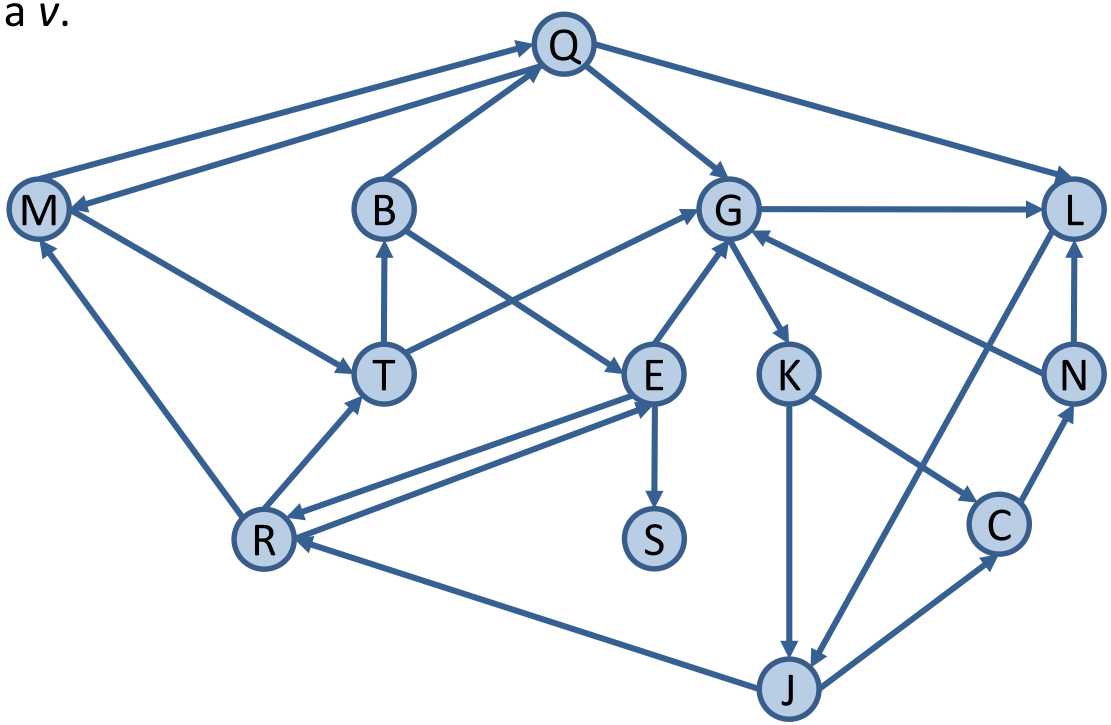
UCA
Universidad de Cádiz

Versión 1.0

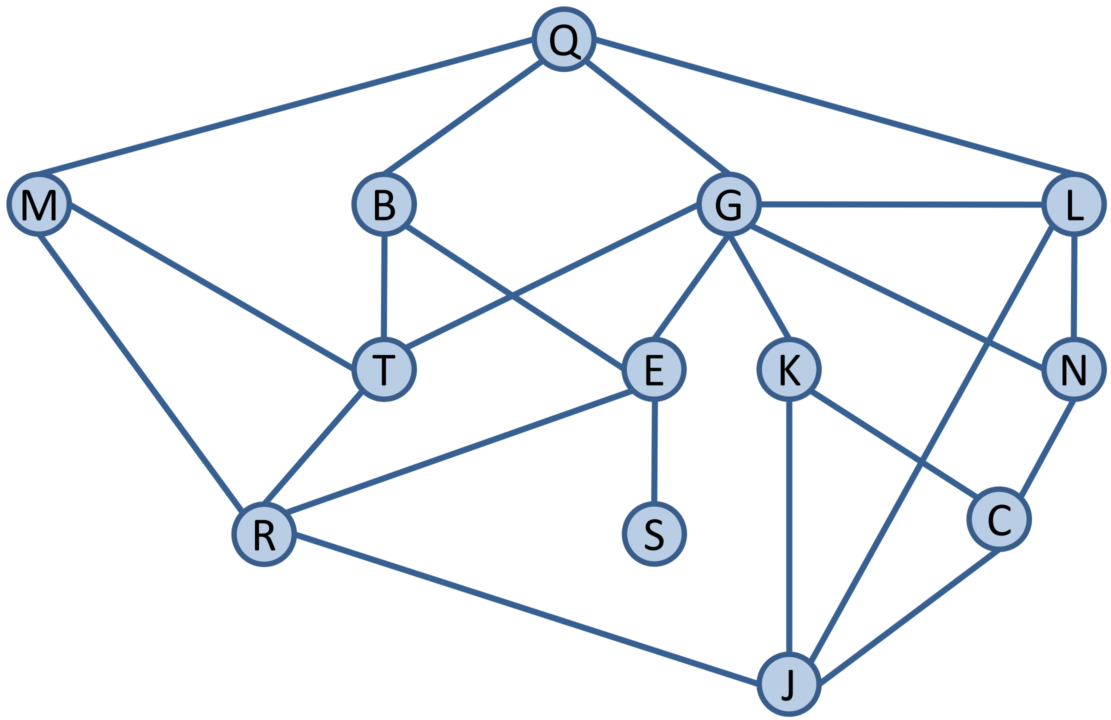
Concepto:

Un ***grafo*** *G = (V, A)* consta de un conjunto de ***vértices*** o ***nodos***, *V*, y un conjunto de ***aristas*** o ***arcos*** $A \subseteq (V \times V)$ que define una relación binaria en *V*. Cada arista es, por tanto, un par de vértices $(v, w) \in A$.

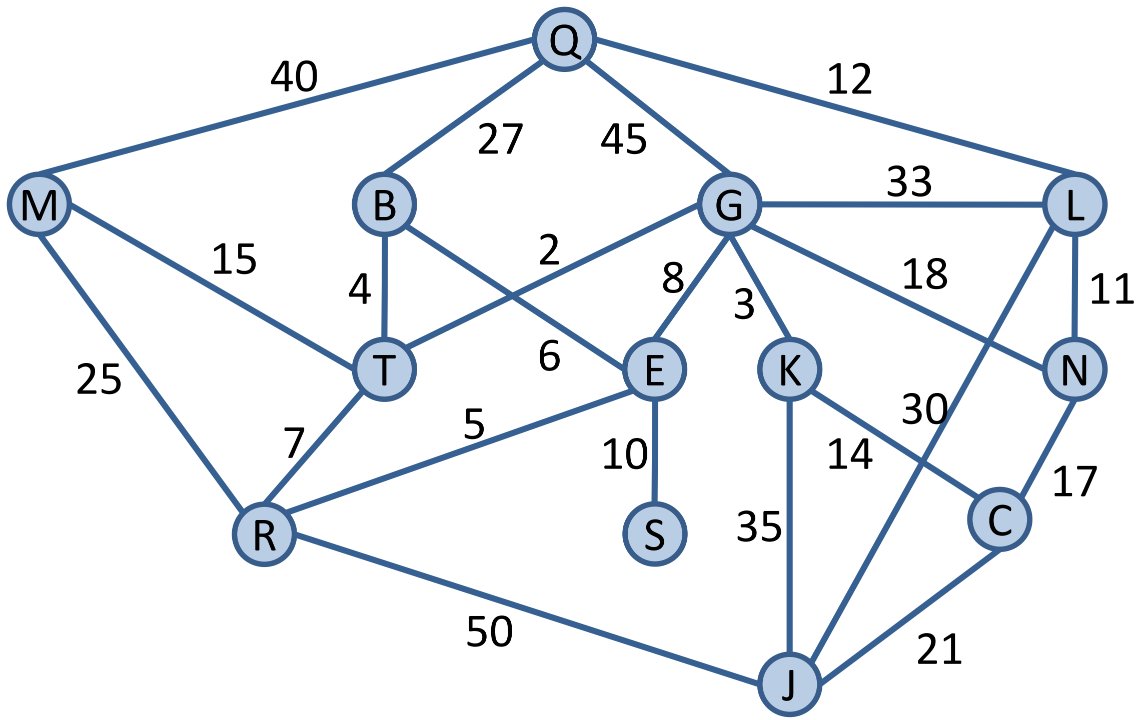
Si cada arista $(v, w) \in A$ es un par ordenado, es decir, si (v, w) y (w, v) no son equivalentes, entonces el grafo es ***dirigido*** y la arista (v, w) se representa como una flecha de *v* a *w*. El vértice *v* se dice que es *incidente* sobre el vértice *w* y *w* es *adyacente* a *v*.



Si, por el contrario, cada arista es un par no ordenado de vértices y por tanto (v, w) = (w, v), entonces el grafo es ***no dirigido*** y la arista (v, w) se representa como un segmento entre v y w. En este caso, se dice que v y w son *adyacentes* y la arista (v, w) es incidente sobre v y w.



Si, por el contrario, cada arista es un par no ordenado de vértices y por tanto (v, w) = (w, v), entonces el grafo es ***no dirigido*** y la arista (v, w) se representa como un segmento entre v y w. En este caso, se dice que v y w son *adyacentes* y la arista (v, w) es incidente sobre v y w.



Una arista puede tener un valor asociado, llamado peso, que representa un tiempo, una distancia, un coste, etc. Un grafo cuyas aristas tienen pesos asociados recibe el nombre de grafo ***ponderado***.

Definiciones

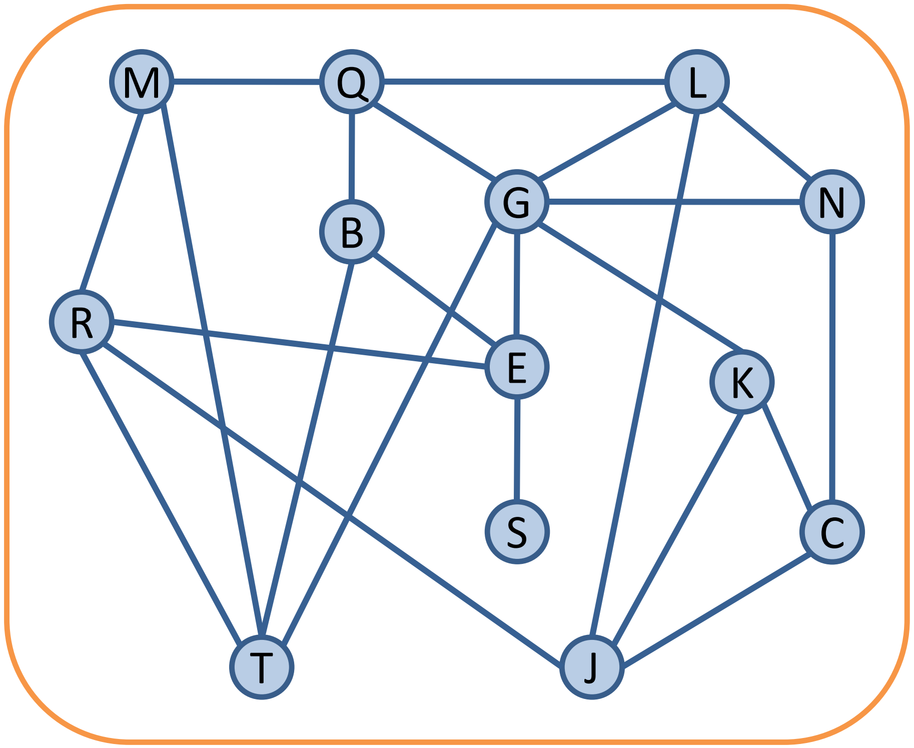
Grado: El grado de un vértice en un grafo no dirigido es el número de arcos del vértice. Si el grafo es dirigido, se distingue entre *grado de entrada* (número de arcos incidentes en el vértice) y *grado de salida* (número de arcos adyacentes al vértice).

Camino: Una sucesión de vértices de un grafo $n_1, n_2, \ldots, n_k$, tal que (n_i, n_{i+1}) es una arista para $1 \leq i < k$. La *longitud* de un camino es el número de arcos que comprende, en este caso k-1. Si el grafo es ponderado la longitud de un camino se calcula como la suma de los pesos de las aristas que lo constituyen.

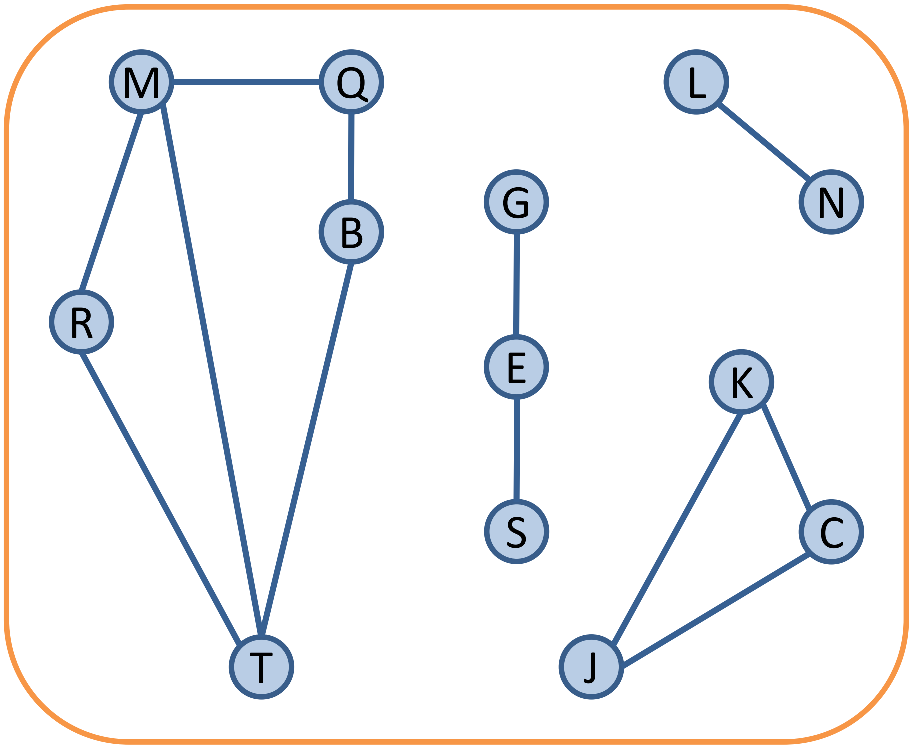
Camino simple: Un camino cuyos arcos son todos distintos. Si además todos los vértices son distintos, se llama *camino elemental*.

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 se llama *elemental*. Se permiten arcos de un vértice a sí mismo; si un grafo contiene arcos de la forma (v, v), lo cual no es frecuente, estos son ciclos de longitud 1 (o del peso asociado al arco); de lo contrario y como caso especial, un vértice v por sí mismo denota un camino de longitud 0.

Grafo conexo: Grafo no dirigido en el que hay al menos un camino entre cualquier par de vértices.

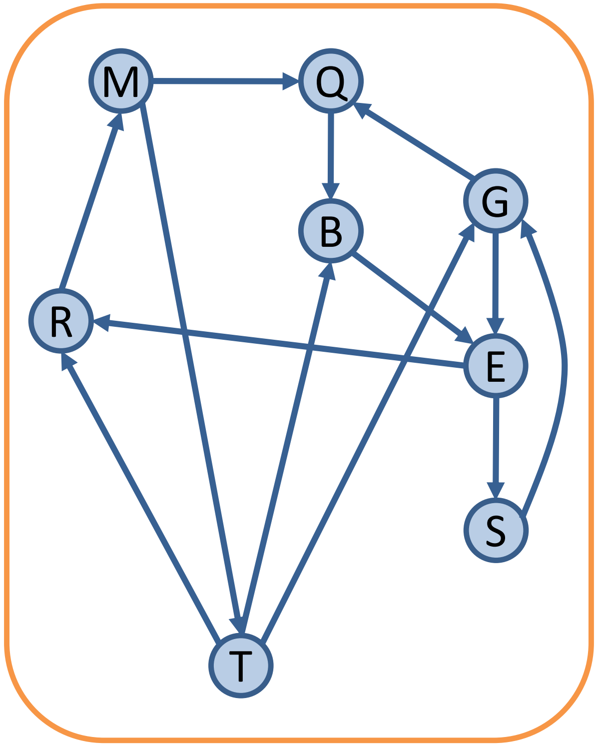


Grafo conexo

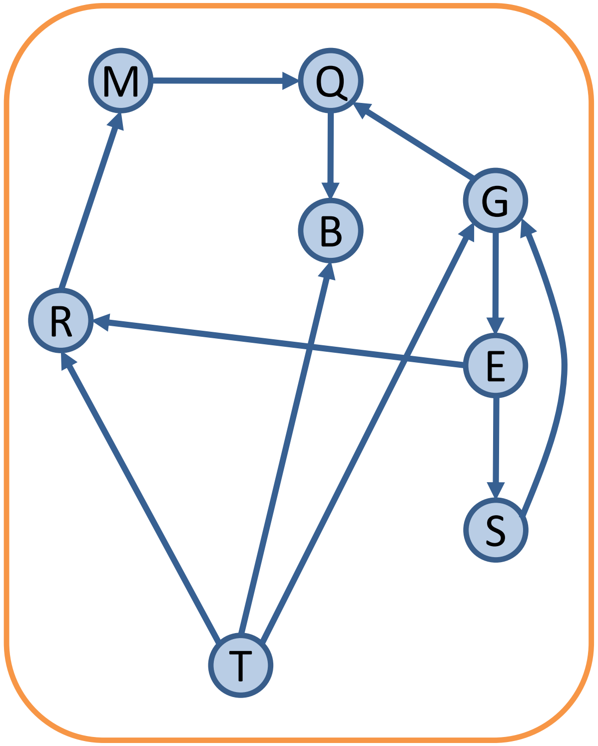


Grafo no conexo

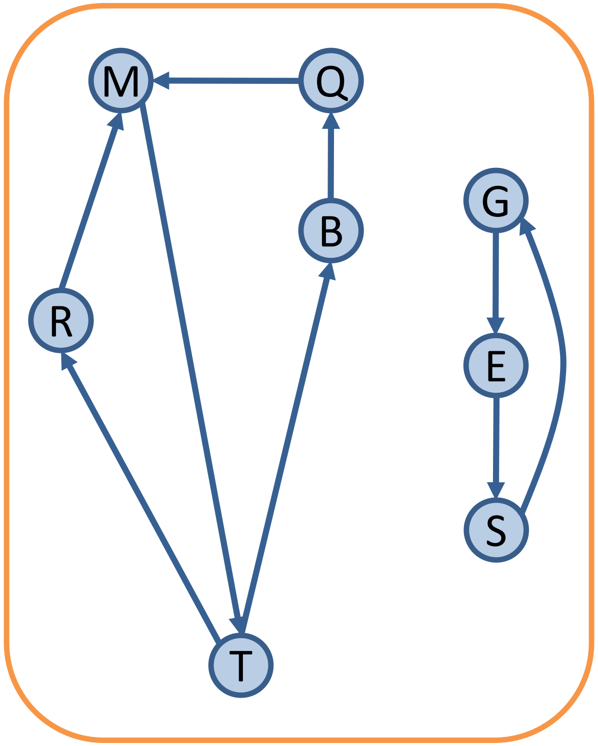
Grafo fuertemente conexo: Grafo dirigido en el que hay al menos un camino entre cualquier par de vértices. Si un grafo dirigido no es fuertemente conexo, pero el grafo no dirigido subyacente (sin dirección en los arcos) es conexo, entonces es *débilmente conexo*.



Grafo fuertemente conexo

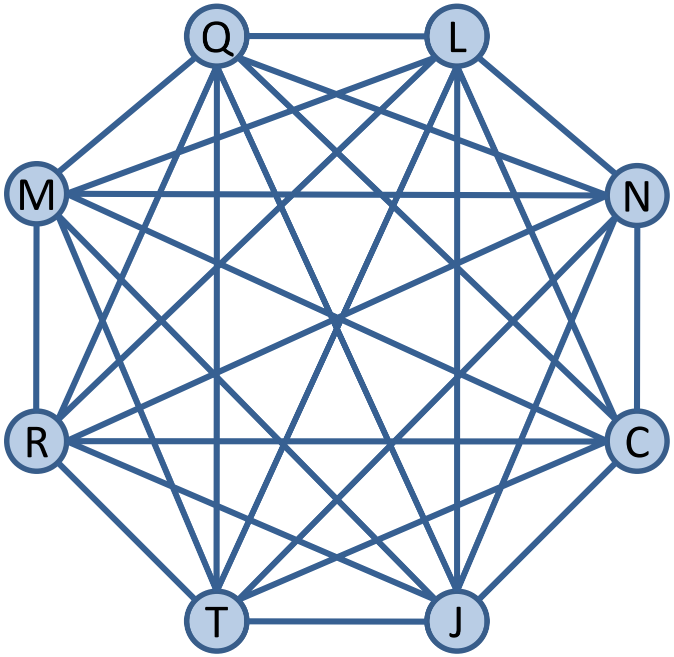


Grafo débilmente conexo

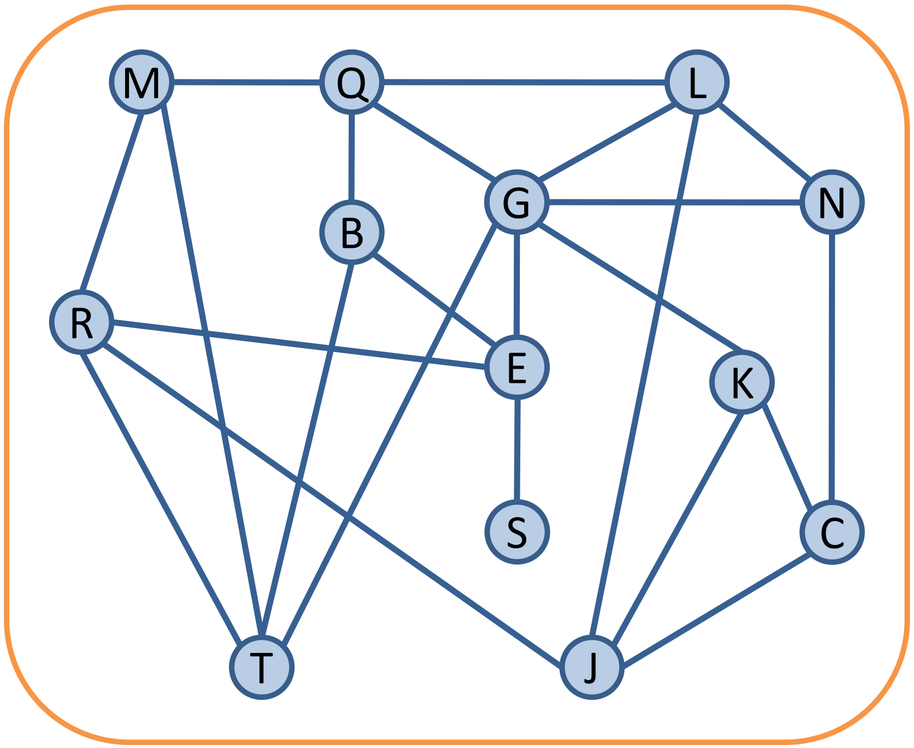


Grafo no conexo

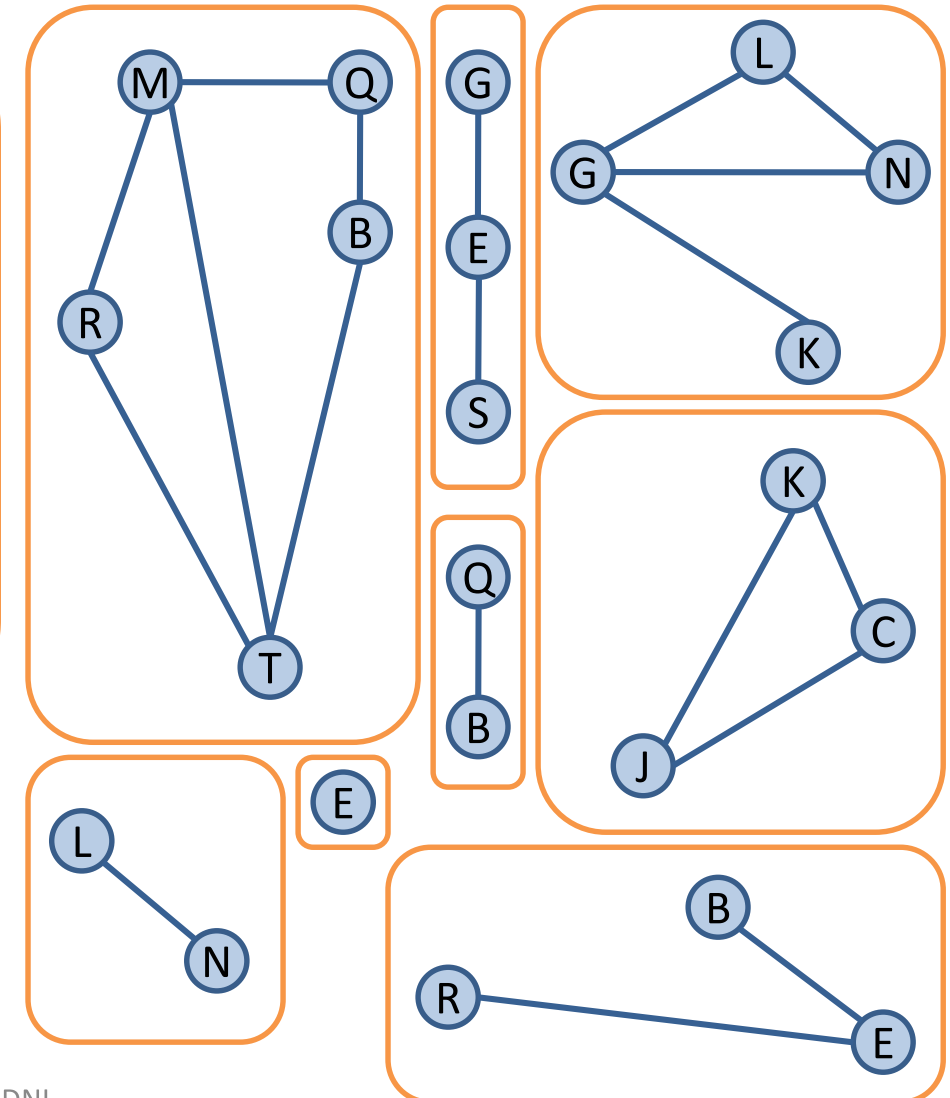
Grafo completo: Aquel en el cual existe una arista entre cualquier par de vértices (en ambos sentidos si el grafo es dirigido).



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'* sólo contiene todas las aristas de *A* que unen vértices de *V'*.

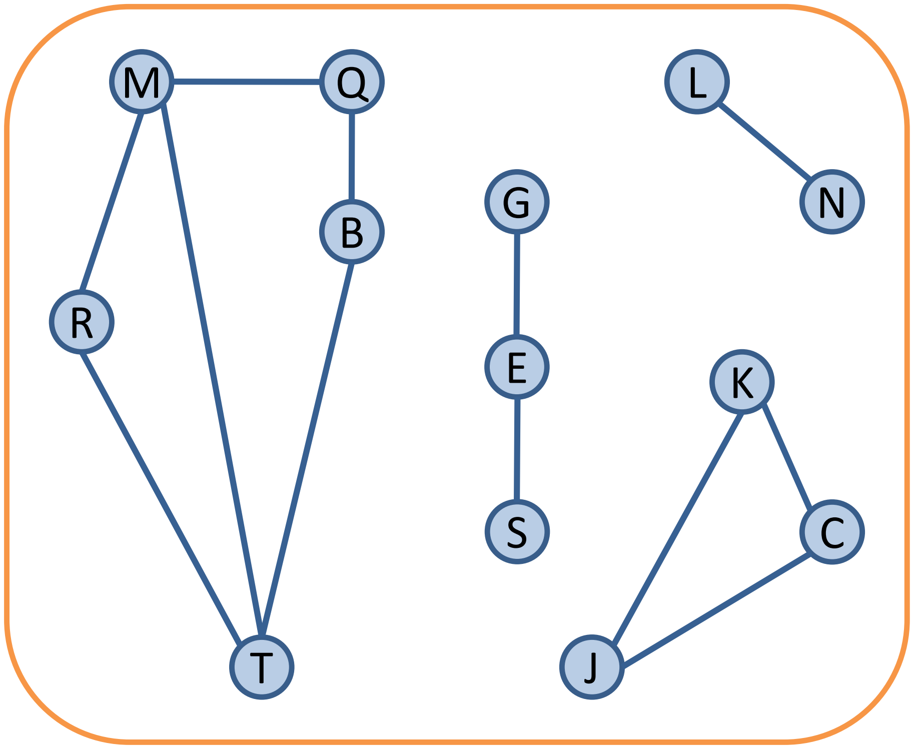


Grafo G

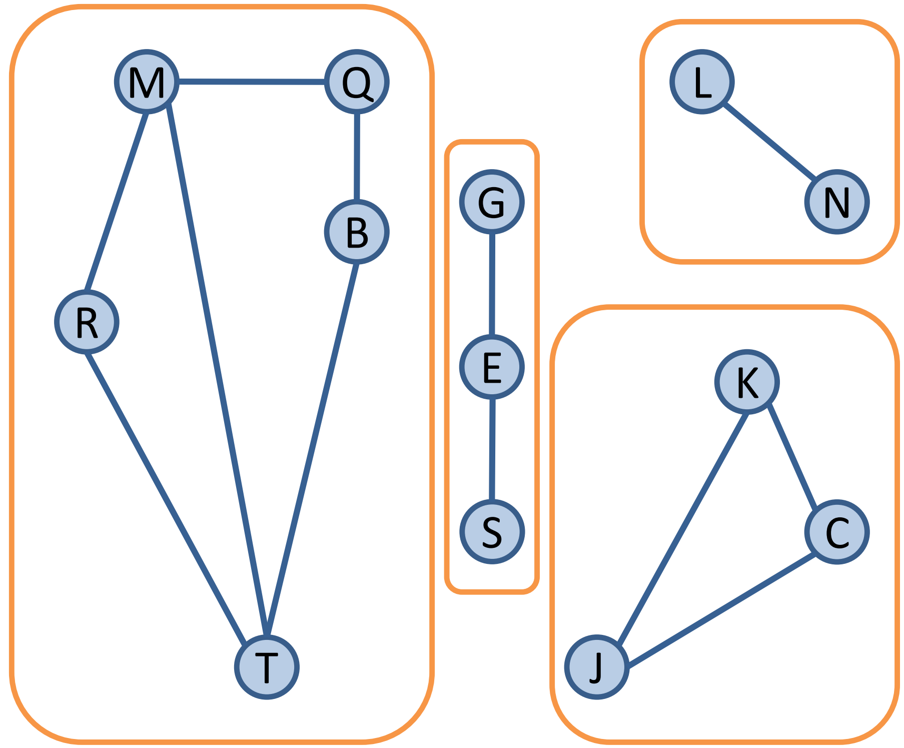


Algunos subgrafos de G

Un **componente conexo** de un grafo no dirigido *G* es un subgrafo conexo maximal, es decir, un subgrafo conexo que no es subgrafo de ningún otro subgrafo conexo de *G*. Análogamente se define *componente fuertemente conexo* de un grafo dirigido.



Grafo G



Componentes conexos de G

Representaciones de grafos

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 $M_{n \times n}$ donde

$M_{i,j} = 1$ si $(i, j) \in A$ y $M_{i,j} = 0$ si $(i, j) \notin A$

Si G es un grafo no dirigido, M es una matriz simétrica ya que $(i, j) = (j, i)$ para cualesquiera vértices i, j.

```
#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;
};
```

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 $C_{n \times n}$ donde

$C_{i,j} = p$ si $(i, j) \in A$, donde p = peso asociado a (i, j)

$C_{i,j} = peso\_ilegal$ si $(i, j) \notin A$, ($peso\_ilegal$ = valor no válido como peso de un arco)

```
#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;
};
```

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

3. Listas de adyacencia:

Asociamos a cada vértice *i* del grafo una lista que almacena todos los vértices adyacentes a *i*.

3.1. Grafos no ponderados:

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

class Grafo {
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 vértices
};
```

3.2. 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 {  // vértice 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 vértice-coste
};
```

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

Representaciones de grafos

Ventajas e inconvenientes:

- Las matrices de adyacencia y costes son muy eficientes para comprobar si existe una arista entre un vértice y otro.
- Pueden desaprovechar gran cantidad de memoria si el grafo no es completo.
- 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.
- 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.