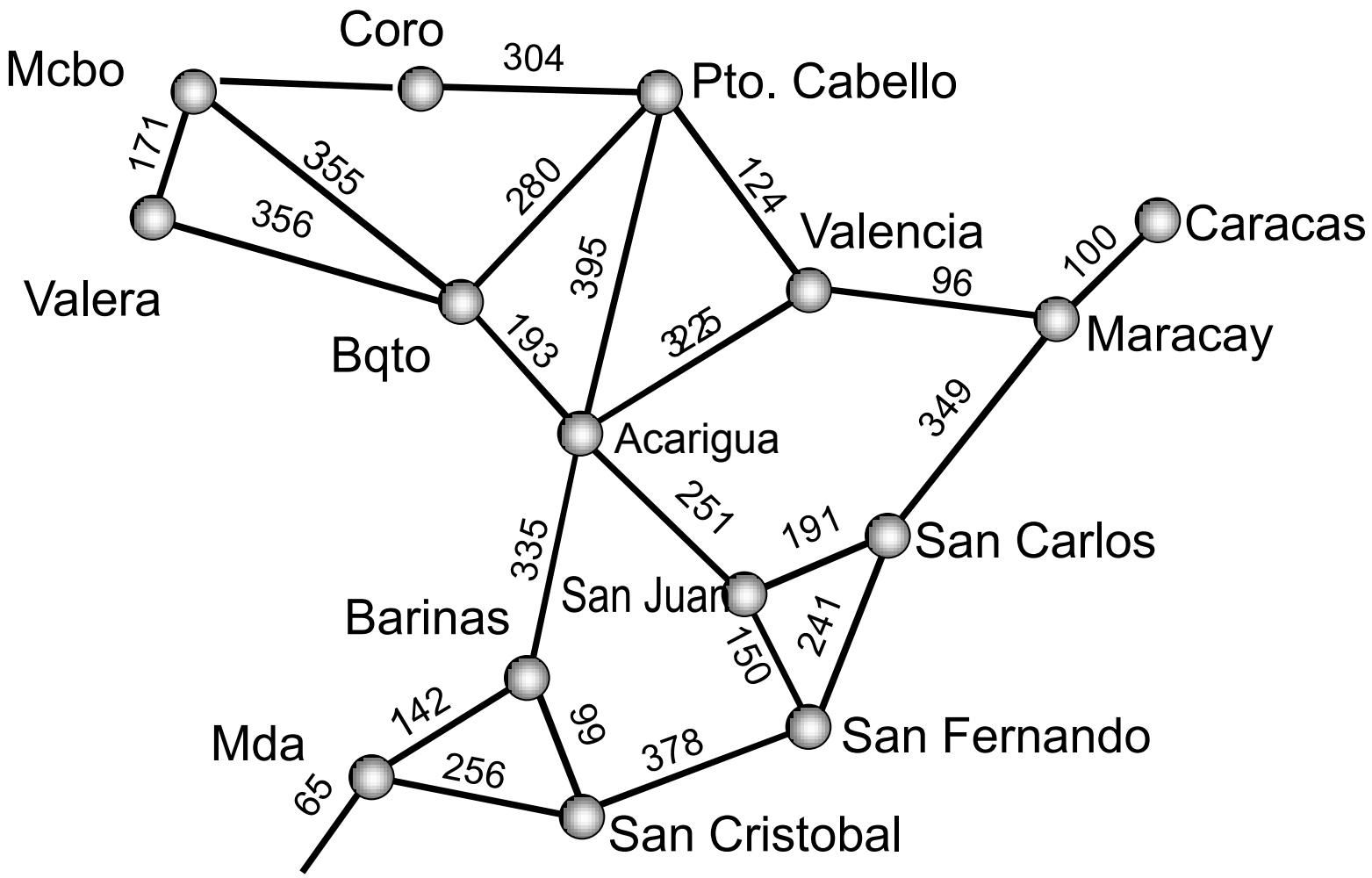
# **Árboles abarcadores mínimos:** algoritmo de Prim y algoritmo de Kruskal.

**Jose Aguilar**

# Grafo de carreteras entre ciudades

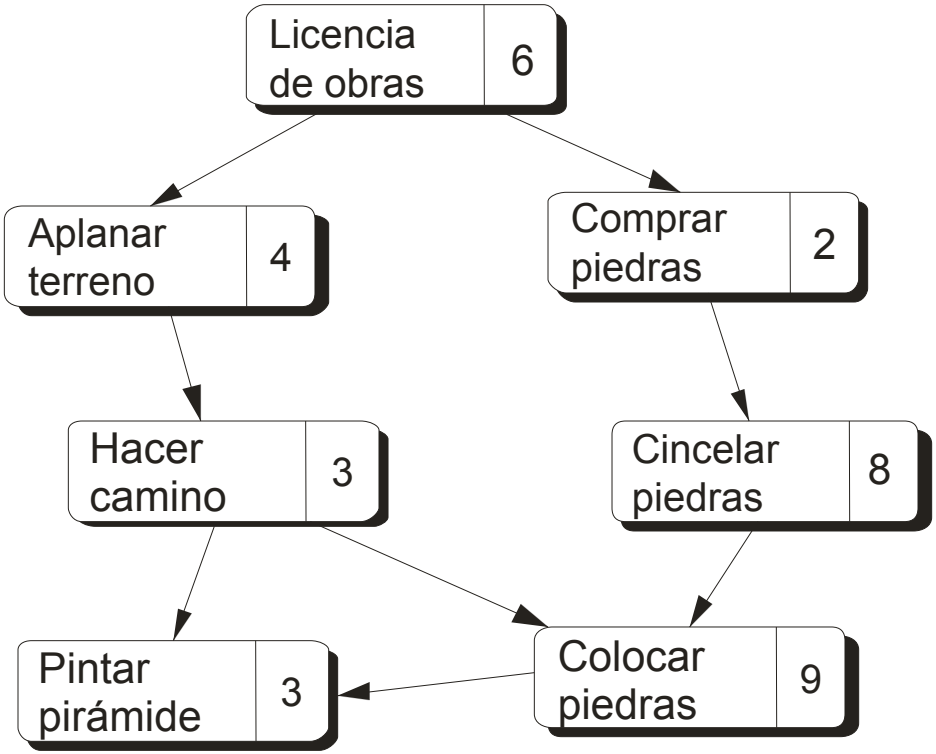


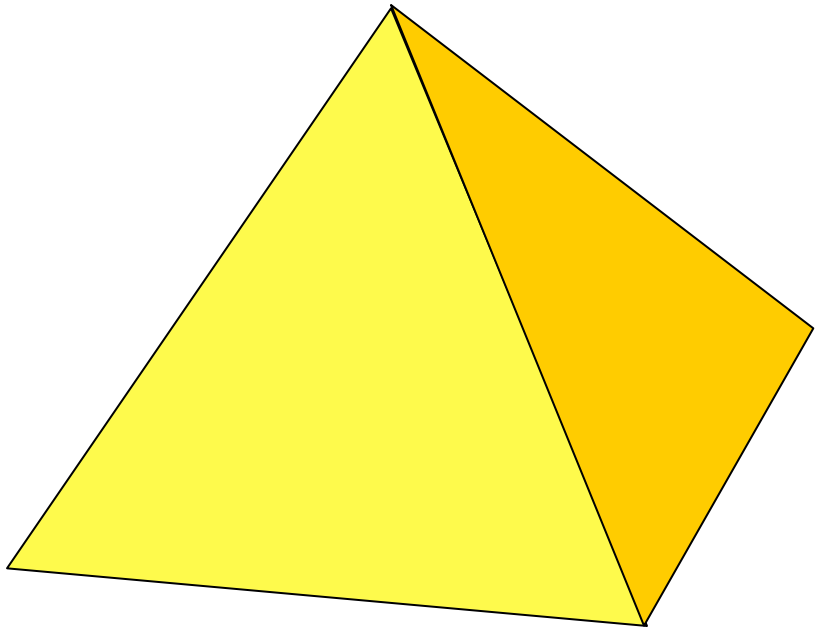
# Grafo de carreteras entre ciudades

## Problemas

- ¿Cuál es el camino más corto de Valera a Ccs?
- ¿Existen caminos entre todos los pares de ciudades?
- ¿Cuál es la ciudad más lejana a Valencia?
- ¿Cuál es la ciudad más céntrica?
- ¿Cuántos caminos distintos existen de Mda a Barinas?
- ¿Cómo hacer un tour entre todas las ciudades en el menor tiempo posible?

# Grafo de planificación de tareas



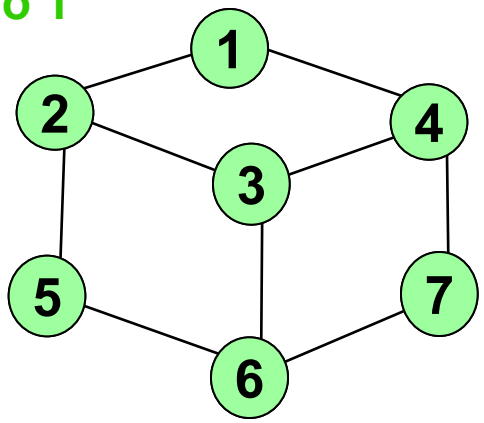
# Grafo de planificación de tareas

**Problemas**

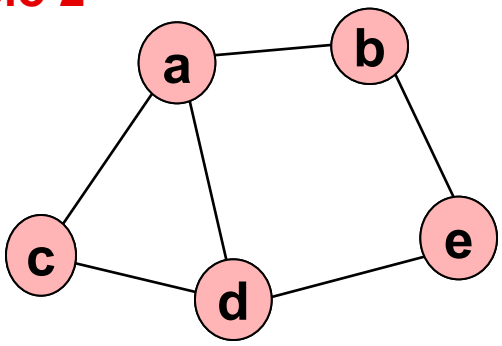
- ¿En cuanto tiempo, como mínimo, se puede construir la pirámide?
- ¿Cuándo debe empezar cada tarea en la planificación óptima?
- ¿Qué tareas son más críticas (es decir, no pueden sufrir retrasos)?

# Grafos asociados a dibujos

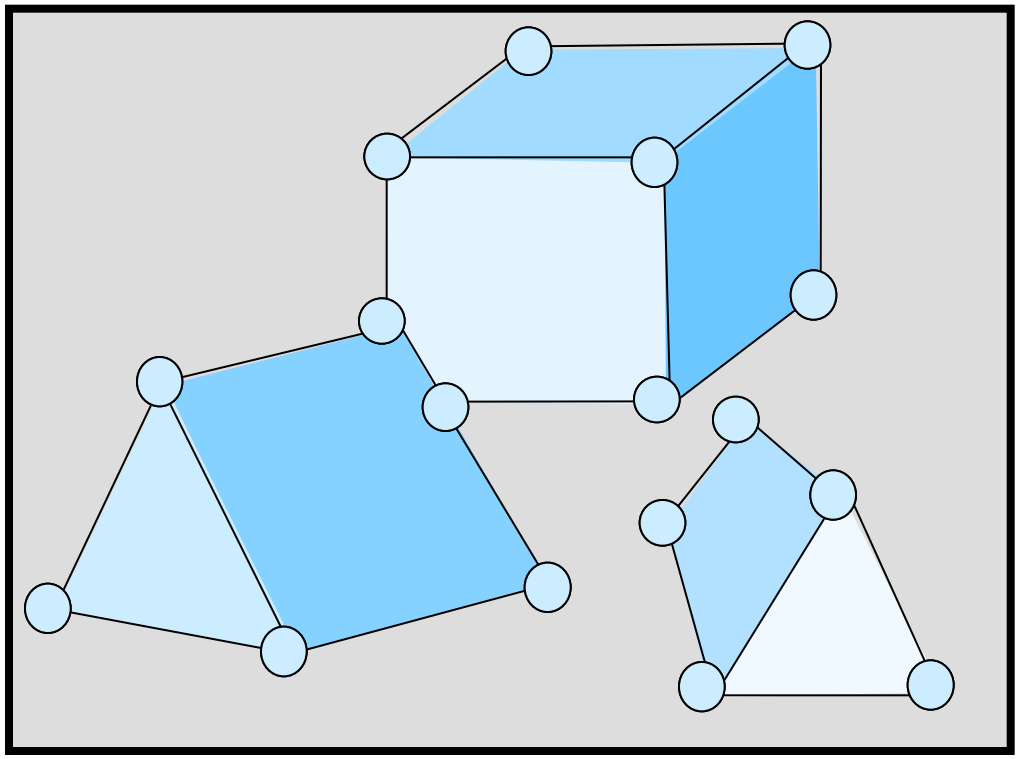
**Modelo 1**




**Modelo 2**




**Escena**



# Grafos asociados a dibujos

## Problemas

- ¿Cuántos grupos hay en la escena?
- ¿Qué objetos están visibles en la escena y en qué posiciones?
- ¿Qué correspondencia hay entre puntos del modelo y de la escena observada?

# Árboles Expansión Mínimos

- En ocasiones se presenta el problema de elegir uno de varios **árboles de expansión que cumplan con el requisito de que la suma total del peso de sus vértices sea la mínima posible.**

- Este es un problema de optimización en donde se busca **reducir el costo total de unir una serie de puntos en un grafo,**

por ejemplo, **unir con caminos un conjunto de ciudades de tal forma que la longitud total de los caminos a construir sea el mínimo y que además permita que todas estén conectadas**.

# Árboles de expansión mínimos.

- **Definición:** Un **árbol de expansión** de un grafo G=(V, A) no dirigido y conexo

  es un subgrafo G'=(V, A') conexo y sin ciclos.

- **Ejemplo:** los árboles de expansión en profundidad y en anchura de un grafo conexo.

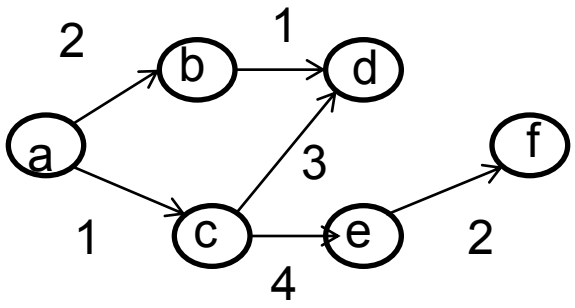
- En grafos con pesos, el **coste del árbol de expansión** es la suma de los costes de las aristas.

- **Problema del árbol de expansión de coste mínimo:**

  Dado un grafo ponderado no dirigido, encontrar el árbol de expansión de menor coste.

# Grafos etiquetados

Es un grafo que tiene un valor entero o real asignado a cada arista




**Un grafo etiquetado es un grafo G = (N, A) donde sus aristas tienen asignada alguna información.**

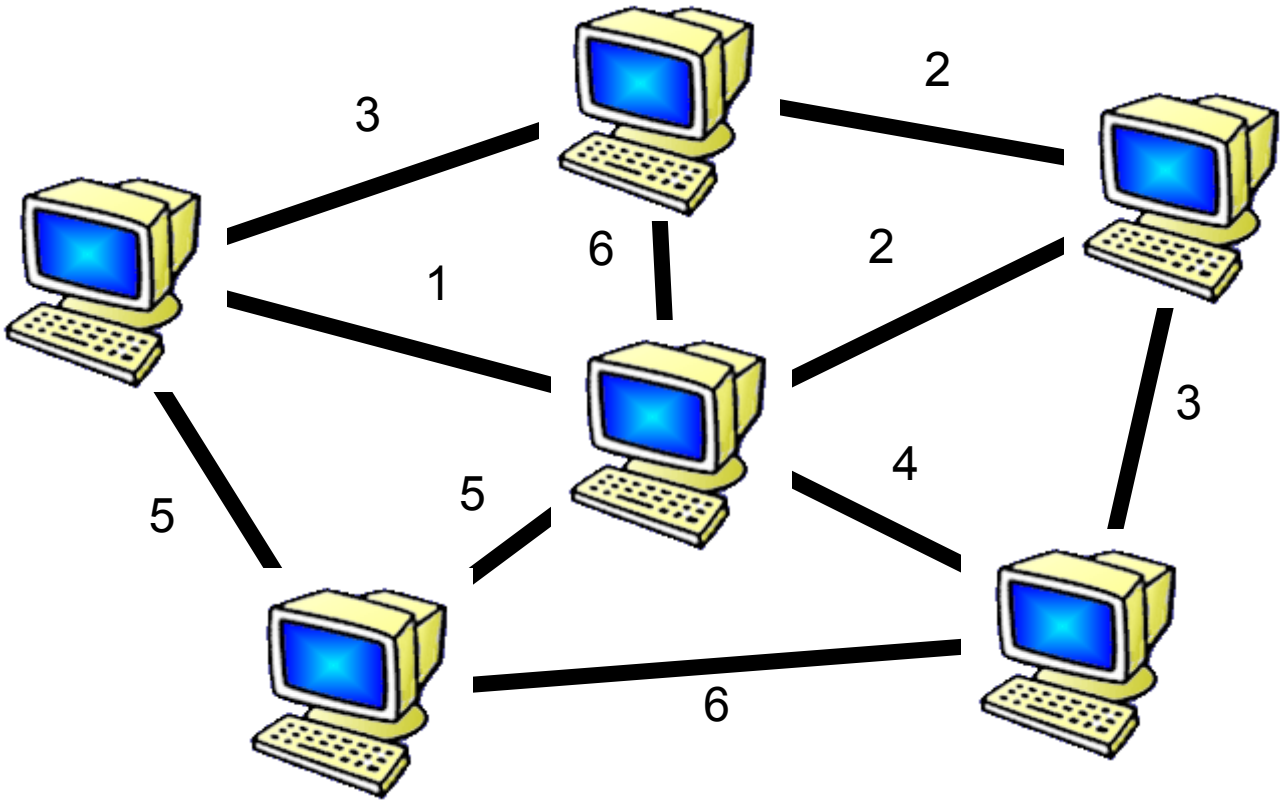
# Árboles de expansión o recubrimiento mínimo

Sea un grafo etiquetado no dirigido y conexo G = (N, A), el árbol de expansión (T) de G es un subconjunto acíclico de A,

$$T \subset A, w(T) = \text{donde } w: A \rightarrow \Re.$$

**Encontrar T es el denominado *problema del árbol de expansión mínimo.***

# Árboles de expansión mínimos.




- **Problema:** Conectar todos los ordenadores con el menor coste total.
- **Solución:** Algoritmos clásicos de Prim y Kruskal (O(AElgV))

Usos de los AEM:

- Identificación de grupos en un conjunto de puntos.
- Grafos esparcidos que dan bastante información del grafo original.
- Cableado de la compañía telefónica, cable, etc.

# ALGORITMO DE PRIM

- El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima. Esto significa que en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol.

- El árbol recubridor mínimo está completamente construido cuando no quedan más vértices por agregar.

# Algoritmo de Prim

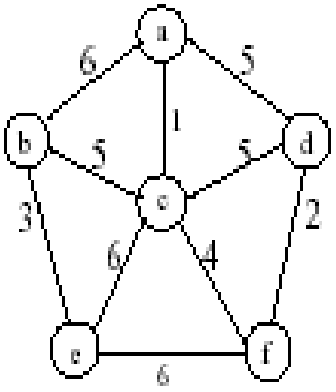
- Se mantienen dos conjuntos, el de los vértices incluidos en el árbol y el de los que no lo están.
- El procedimiento consiste en elegir la arista de menor peso que une un vértice en el conjunto del árbol con un vértice que no esta en el árbol.
- Es un algoritmo incremental.
- El árbol formado es un árbol simple, que se comienza a formar con un nodo arbitrario y crece hasta tener todos los nodos en X.
- Se implementa con una **cola por prioridad** para seleccionar fácilmente la nueva arista a ser incluida en X.

# Algoritmo de Prim

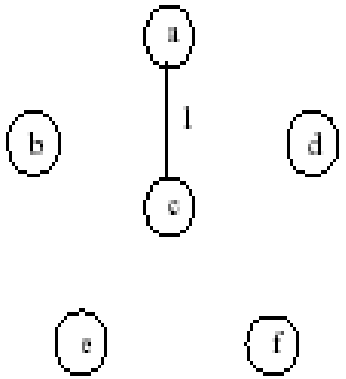
**La idea básica consiste en añadir, en cada paso, una arista de peso mínimo a un árbol previamente construido**

1. Empezar en un vértice cualquiera **v**. El árbol consta inicialmente sólo del nodo **v**.
2. Del resto de vértices, buscar el que esté más próximo a **v** (es decir, con la arista (**v**, **w**) de coste mínimo). Añadir **w** y la arista (**v**, **w**) al árbol.
3. Buscar el vértice más próximo a cualquiera de estos dos. Añadir ese vértice y la arista al árbol de expansión.
4. Repetir sucesivamente hasta añadir los **n** vértices.

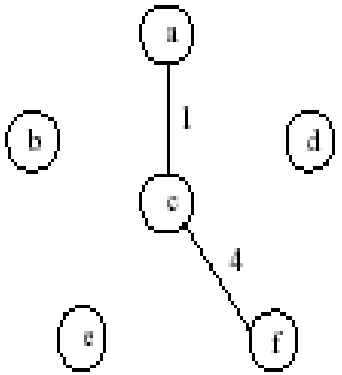
```
void Prim( GRAFO G, CONJUNTO T )
{
   CONJUNTO_V  U;
   VERTICE     v;

   T = ∅;
   U = { cualquier vértice de G };
   while( U != V ) {
     Sea (u,v) es el arco de menor costo tal que
         u ∈ U  y  v ∈ V-U;
     T = T ∪ { (u,v) };
     U = U ∪ { v };
   }
}
```



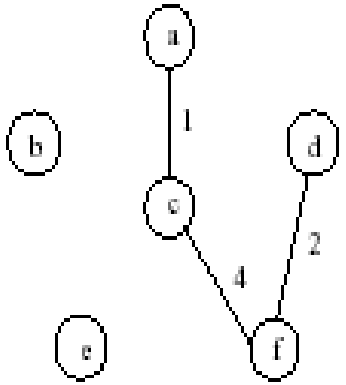

Grafo original



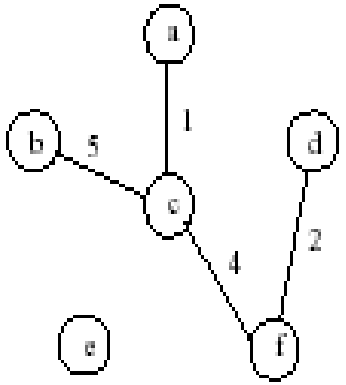

a)



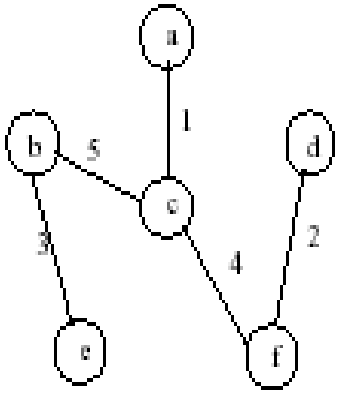

b)




c)




d)




e)

# Algoritmo de Prim

Versión 1.0

PrimAEM(Nodo: r, Arreglo[n]De Nodo: &clave): Arreglo[n]De Nodo

{pre: n > 0 ∧ r ∈ N } {pos: n > 0 ∧ G' = G ∧ }

| | | |
|---|---|---|
| 1 | [ c.entrar(v, MV) ] v ∈ N | -c. ColaPrioridad[X]. Cola de nodos. |
| 2 | clave( r ), padre( r ) = 0, Nulo | -clave: Arreglo[n]De Nodo. Variable auxiliar |
| 3 | ( ¬c.vaciaCola( ) )[ u = c.min( ) | con los pesos. |
| 4 | [Si (v ∈ c ∧ w(u, v) < clave( v )) ent. | -entrar(), vaciaCola(), min(). Definidas |
| | padre( v ) = u | en la clase ColaPrioridad. |
| | clave( v ) = w(u, v) | -padre. Arreglo[n]De Nodo. Variable auxiliar |
| | fsi ] ∀ v ∈ u.listaAdyacencia | con el árbol de expansión mínima. |
| | regrese padre | |

$$T(n) = O(A + N \lg N)$$

# Algoritmo de Prim

PRIM(G, r)
para cada vértice u en G
    clave[u] = infinito
    padre[u] = NULO
clave[r] = 0
Meter los vértices  u de G a una **cola de prioridad** Q con clave[u]
Mientras no este vacía Q
Extraer un vértice de Q y llamarlo u
Para cada vértice v que sea adyacente a u
    Si v esta en Q y el peso de (u,v) < clave[v]
        padre[v] = u
        clave[v] = w(u,v)

La clase ColaPrioridad debe ser implantada con montículos de Fibonacci para poder tener un tiempo mejor que el algoritmo de Kruskal. Si se implanta con un montículo binario su complejidad es igual a la del algoritmo de Kruskal.

# Colas por prioridad

**Características:**

- Su nombre viene de una aplicación de Sistemas Operativos: el mantenimiento de las colas internas de procesos, donde esos procesos son manejados según su prioridad asignada.
- Cada entrada en la cola es un par [clave, valor]
- Clave: es un campo especial para reconocer la entrada. Las claves están siempre ordenadas según un orden total
- Los valores asociados a las claves se pueden actualizar, pero no las claves

# Colas por prioridad

Las operaciones más importantes en un TDA de colas por prioridad se refieren aquellas que permiten repetidamente seleccionar el elemento de la cola de prioridad que tiene como clave el valor mínimo (máximo).

Una cola por prioridad P debe soportar las siguiente operaciones:

inserta(ent)
min()
extMin()

crea()
union()
destruye()

# Cola de Prioridad

Implementaciones de un TDA de Cola de Prioridad

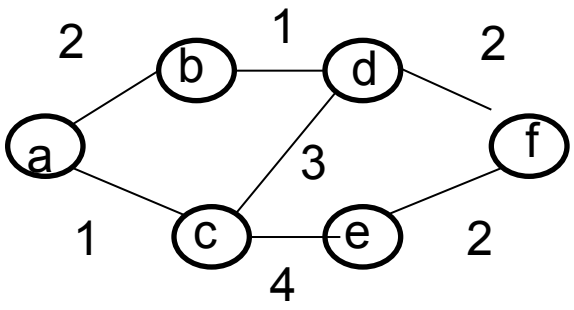
- Arboles equilibrados
- Montículos Binarios
- Montículos a la izquierda
- Montículos oblicuos
- Colas binomiales, colas binomiales perezosas
- Colas de Fibonacci

Un **montículo binario** (o simplemente montículo) es un árbol binario semicompleto en el que el valor de la clave almacenada en cualquier nodo es menor o igual que los valores claves de sus hijos

Propiedad de ordenamiento parcial

# Algoritmo de Prim

| Paso | (u, v) | B |
|---|---|---|



# Algoritmo de Kruskal

- El algoritmo de Kruskal basa su funcionamiento en la elección de las aristas de menor peso que no forman ciclos,

- para poder elegir dichas aristas es necesario usar un método de almacenamiento que las ordene de menor a mayor peso, pero además, de otros artificios matemáticos.

# ALGORITMO DE KRUSKAL

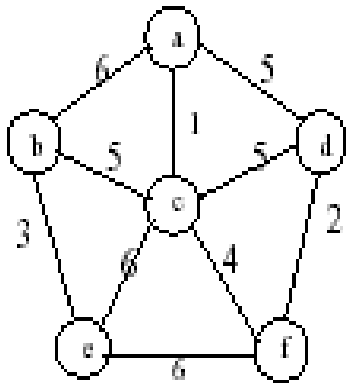
- El algoritmo de Kruskal permite hallar el árbol minimal de cualquier grafo valorado (con capacidades). Hay que seguir los siguientes pasos:
  1. Se marca la arista con menor valor. Si hay más de una, se elige cualquiera de ellas.
  2. De las aristas restantes, se marca la que tenga menor valor, si hay más de una, se elige cualquiera de ellas.
  3. Repetir el paso 2 siempre que la arista elegida no forme un ciclo con las ya marcadas.
  4. El proceso termina cuando tenemos todos los nodos del grafo en alguna de las aristas marcadas, es decir, cuando tenemos marcados *n-1* arcos, siendo *n* el número de nodos del grafo

# ALGORITMO DE KRUSKAL

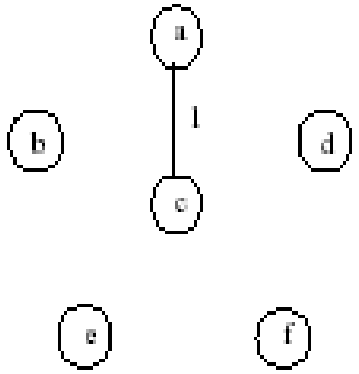
**Esquema: G= (V, A)**

1. Empezar con un grafo sin aristas: **G'**= (V, Ø)
2. Seleccionar la arista de menor coste de **A**.
3. Si la arista seleccionada forma un ciclo en **G'**, eliminarla. Si no, añadirla a **G'**.
4. Repetir los dos pasos anteriores hasta tener **n-1** aristas.

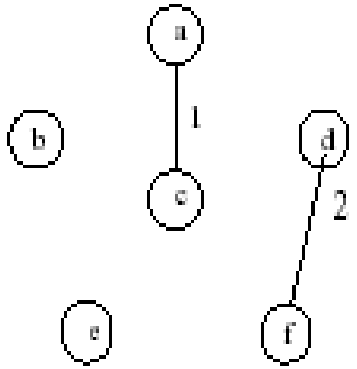
- ¿Cómo saber si una arista **(v, w)** provocará un ciclo en el grafo **G'**?



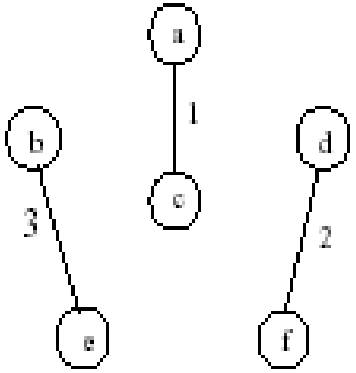

Grafo original



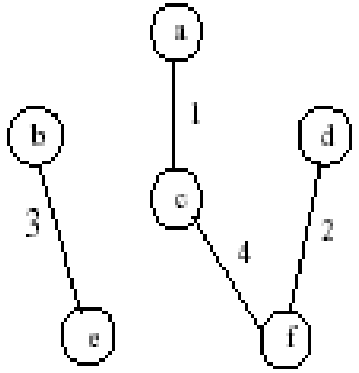

a)



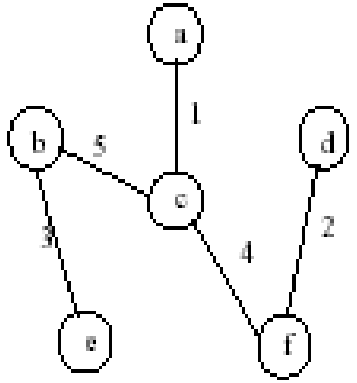

b)




c)




d)




e)

# ALGORITMO DE KRUSKAL.

- **Necesitamos:**
  - Ordenar las aristas de **A**, de menor a mayor: **O(a log a).**
  - Saber si una arista dada **(v, w)** provocará un ciclo.
- ¿Cómo comprobar rápidamente si **(v, w)** forma un ciclo?
- Una arista **(v, w)** forma un ciclo si **v** y **w** están en el mismo componente conexo.
- La relación “estar en el mismo componente conexo” es una **relación de equivalencia**.

# ALGORITMO DE KRUSKAL

- Usamos la estructura de **relaciones de equivalencia** con punteros al padre:

  - Inicialización: crear una relación de equivalencia vacía
  - Seleccionar las aristas **(v, w)** de menor a mayor.
  - La arista forma ciclo si: **Encuentra(v)**=**Encuentra(w)**
  - Añadir una arista **(v, w)**: **Unión(v, w)** (juntar dos componentes conexos en uno).

# Algoritmo

1 A = Φ

2 while A does not form a MST

3 do find an edge (u,v) safe for A

4 A<-A U {(u,v)}

5 return A

# Árboles de expansión o recubrimiento mínimo

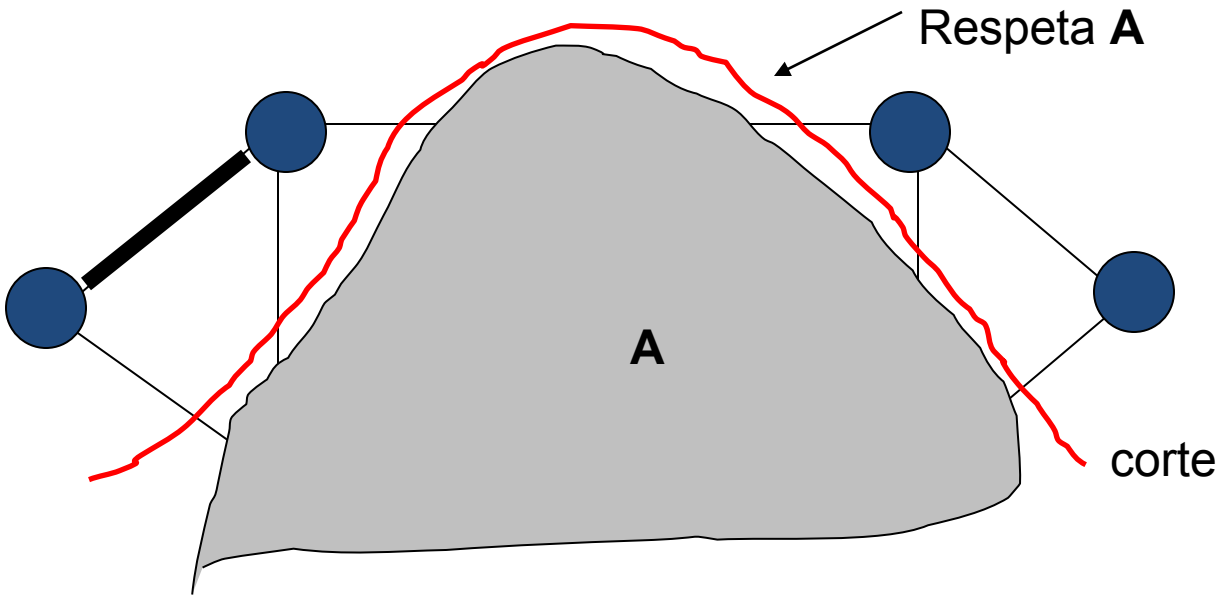
Aplicación:
G={N, A} N={ciudades} y A={costo de línea telefónica del nodo a al nodo b}

**El árbol abarcador mínimo T de G es la red más barata para conectar las ciudades utilizando conexiones directas**

- C={candidatos}=A
- T={solución}
- Conjunto de aristas seguras si no contiene ciclos
- Función objetivo: minimizar la longitud total de las aristas en T

# Arcos Seguros

- **Corte**: (S, V-S) es una partición de V.




**Blancos**: parte del MST.
**Negros**: aún por ser considerados.

**Debe encontrarse un Arco Liviano del corte para incluirlo en la solución**.

# Conceptos

- **Corte:** Un corte (S, N-S) de un grafo no dirigido G = (N, A) es una partición de N.

- Una arista (u, v) ∈ A cruza el corte (S, N-S) si uno de los nodos terminales de la arista está en S y el otro en N-S.

**Un corte respeta A si no hay aristas en A que crucen el corte.**

**Una arista es ligera cruzando el corte si su peso es el mínimo de cualquier arista cruzando el corte.**

- Una arista es ligera satisfaciendo una propiedad dada, si su peso es el mínimo para cualquier arista que satisfaga la propiedad.

# Noción de Corte

**Teorema**:

Sea G = (V,E) un grafo conexo, con

$w : E \rightarrow \mathbf{R}$.

Sea A un subgrafo de E que se incluye en algún MST $\subset$ G.

Sea (S, V-S) cualquier corte de G que respeta A y

Sea (u,v) un **Arco Ligero** que cruza (S,V-S).

*(u,v) es seguro para A*.

# Conceptos

**Teorema:** Sea G =(N, A) un grafo no dirigido conexo etiquetado con una función real para los pesos w definida en A, sea X $\subseteq$ A que está incluido en algún árbol de expansión mínima para G, sea (S, N-S) cualquier corte de G que respete X y sea (u, v) una arista ligera cruzando (S, N-S), entonces la arista (u, v) **es *segura*** para X.

UNIVERSIDAD DE LOS ANDES
MERIDA VENEZUELA

# Algoritmo de Kruskal

Versión 1.0

KruskalAEM( ): Conjunto[Arista]

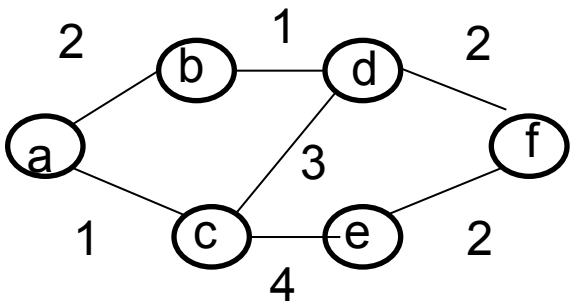
{pre: n > 0 } {pos: n > 0 ∧ G' = G ∧ X es el árbol de expansión mínima de G }

| | | |
|---|---|---|
| 1 | [ cdis.incluir(v) ] v ∈ N | -X. Conjunto[X]. Árbol de expansión mínima resultante para el grafo.<br>-cdis: ConjDisj[TipoClave]. Bosque de nodos del grafo.<br>-incluir( ), buscar( ), union( ). Definidas en la clase ConjDisj.<br>- ∪( ). Función de la clase Conjunto [X]. |
| 2 | Ordene ascendente las aristas de G por sus pesos w | |
| 3 | | |
| 4 | [ Si ( cdis.buscar(v) ≠ cdis.buscar(u) ) entonces<br>X = X ∪ {(u, v)}<br>cdis.union(u, v) // O(A lg A)<br>fsi ] (u, v) ∈ A por orden ascendente de w | |
| | regrese X | |

$$T(n) = O(A \lg A)$$

# Algoritmo de Kruskal

| Paso | arista considerada | componentes conexos |
|---|---|---|



# Árboles de expansión mínimos.

- Ambos algoritmos (**Prim** y **Kruskal**) encuentran siempre la solución óptima.
- La solución obtenida será la misma, o no...
- La estructura de los dos algoritmos es muy parecida:
  - Empezar con una solución “vacía”.
  - Añadir en cada paso un elemento a la solución (Prim: un nodo; Kruskal: una arista).
  - Una vez añadido un elemento a la solución, no se quita (no se “deshacen” las decisiones tomadas).