DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

# Algoritmos greedy sobre grafos

## Análisis y Diseño de Algoritmos

# Algoritmos greedy sobre grafos

- Árboles generadores minimales
  - Algoritmo de Kruskal
  - Algoritmo de Prim
- Caminos mínimos
  - Algoritmo de Dijkstra
- Heurísticas greedy
  - El problema del coloreo de un grafo
  - El problema del viajante de comercio

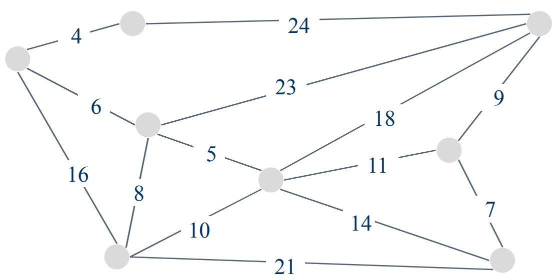
# Árboles generadores minimales

**Problema**
Dado un grafo conexo G = (V, A) no dirigido y ponderado con pesos positivos, calcular un subgrafo conexo $T \subseteq G$ que conecte todos los vértices del grafo G y que la suma de los pesos de las aristas seleccionadas sea mínima.

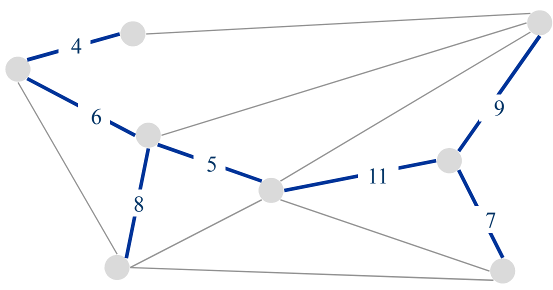
**Solución**
Este subgrafo es necesariamente un árbol: árbol generador minimal o árbol de recubrimiento mínimo (en inglés, "minimum spanning tree" [MST]).



# Árboles generadores minimales




$G = (V, A)$




$T \subset G$

$\Sigma_{a \in T}\, c_a = 50$

# Árboles generadores minimales

**Aplicaciones**

- Diseño de redes: redes telefónicas, eléctricas, hidraúlicas, de ordenadores, de carreteras…

  p.ej. Construcción de redes de mínimo coste
  Refuerzo de líneas críticas
  Identificación de cuellos de botella
  Enrutamiento (evitar ciclos)
  …
- Soluciones aproximadas para problemas NP.
- Algoritmos de agrupamiento (análisis de clusters)
- …



# Árboles generadores minimales

Algoritmos greedy para resolver el problema:

- **Algoritmo de Kruskal**:
  Comenzando con T=∅, considerar las aristas en orden creciente de coste y añadir las aristas a T salvo que hacerlo suponga la creación de un ciclo.

- **Algoritmo de borrado inverso**:
  Comenzando con T=A, considerar las aristas en orden decreciente de coste y eliminar las aristas de T salvo que eso desconectase T.

- **Algoritmo de Prim**:
  Comenzando con un nodo raíz arbitrario s, hacer crecer el árbol T desde s hacia afuera. En cada paso, se añade al árbol T el nodo que tenga una arista de menor coste que lo conecte a otros nodos de T.

## Elementos del algoritmo de Kruskal

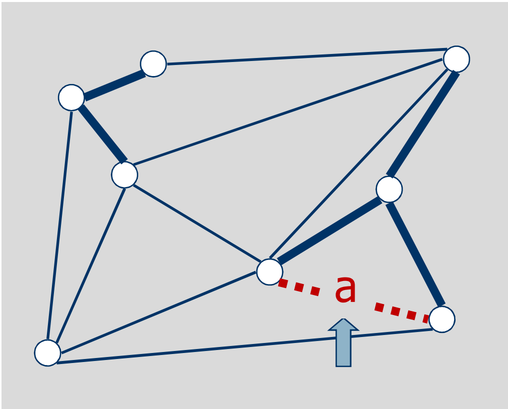
- Conjunto de candidatos: Aristas del grafo.
- Función de selección: La arista de menor coste.
- Función de factibilidad:
  El conjunto de aristas no contiene ningún ciclo.
- Criterio que define lo que es una solución:
  El conjunto de aristas seleccionado conecta todos los vértices (árbol con n-1 aristas).
- Función objetivo: Suma de los costes de las aristas.



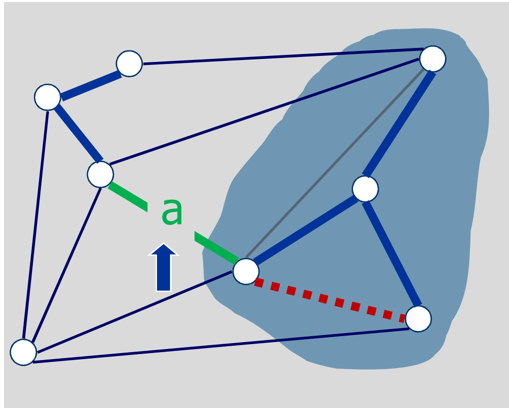


```
función Kruskal( Grafo G(V,A) )
{
  set<aristas> C(A);
  set<aristas> S;              // Solución inicial vacía

  Ordenar(C);

  while (!C.empty() && S.size()!=V.size()-1) {
        x = C.first();         // Arista de menor coste
        C.erase(x);
        if (!HayCiclo(S,x))    // ¿Solución factible?
           S.insert(x);
   }

   if (S.size()==V.size()-1)
      return S;
   else
      return "No hay solución";
}
```

# Árboles generadores minimales
## Algoritmo de Kruskal



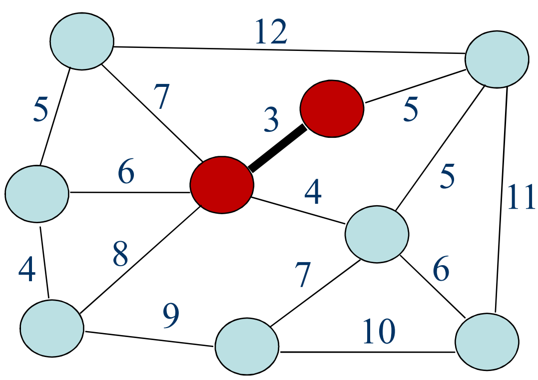

Añadir la arista crearía un ciclo.

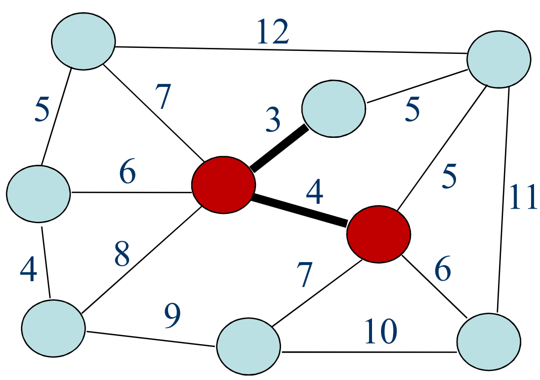



La arista forma parte del árbol generador minimal.

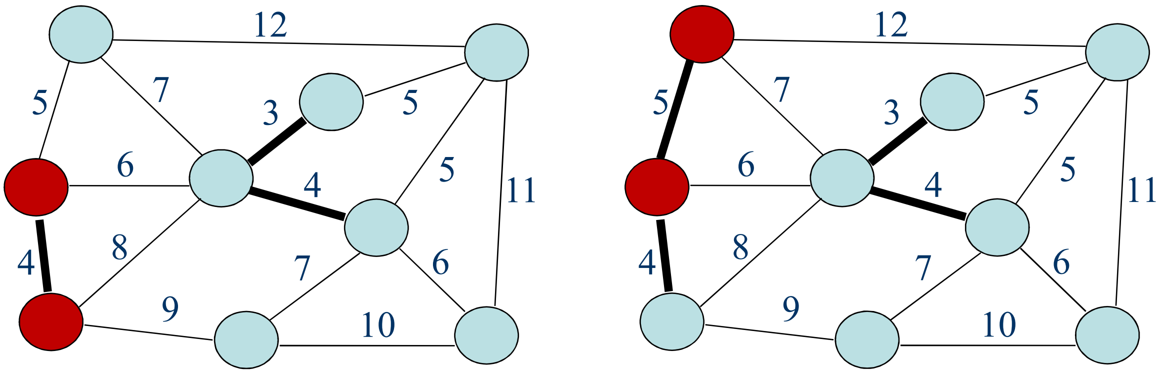


# Árboles generadores minimales
## Algoritmo de Kruskal



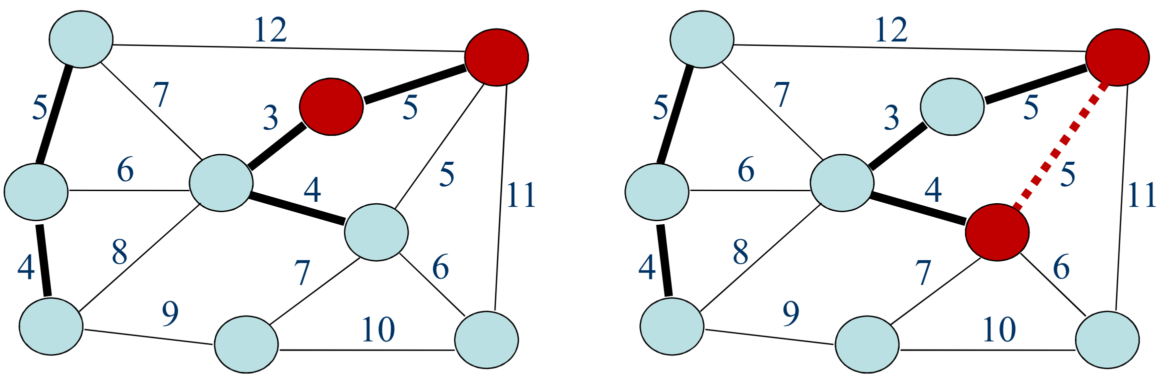



# Árboles generadores minimales
## Algoritmo de Kruskal

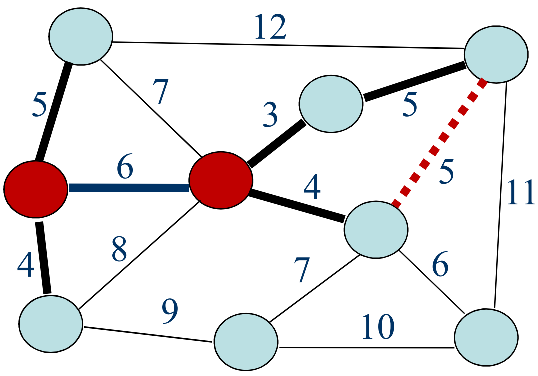


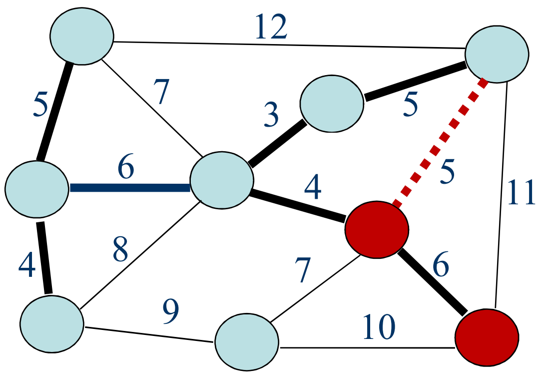



# Árboles generadores minimales
## Algoritmo de Kruskal



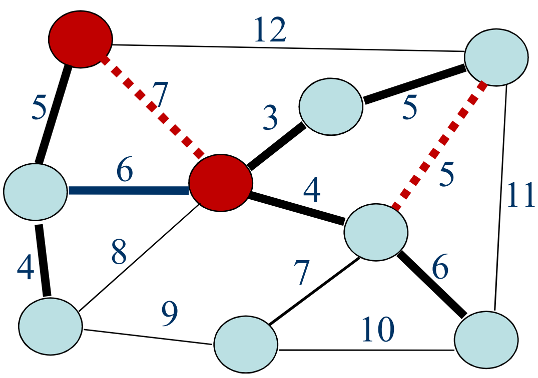
# Árboles generadores minimales
## Algoritmo de Kruskal

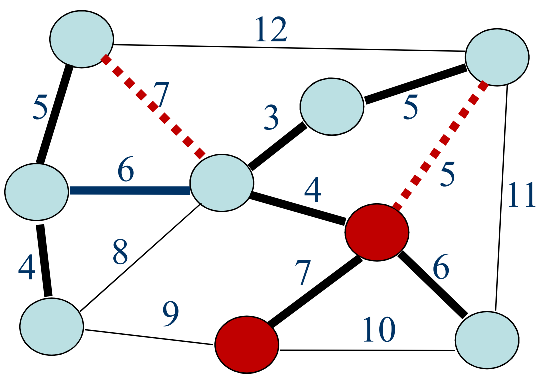


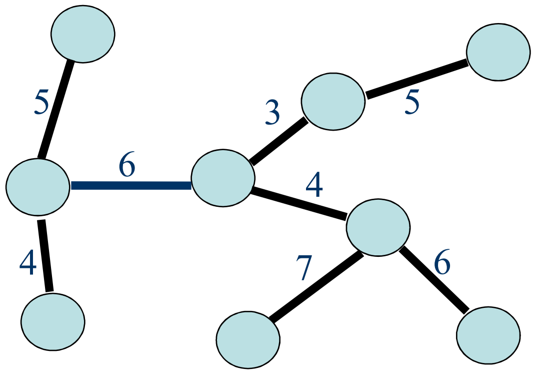



# Árboles generadores minimales
## Algoritmo de Kruskal



# Árboles generadores minimales
## Algoritmo de Kruskal







# Árboles generadores minimales
## Algoritmo de Kruskal


**Optimalidad del algoritmo de Kruskal**

Teorema:

El algoritmo de Kruskal halla un árbol generador minimal.

Demostración:

Por inducción sobre el número de aristas que se han incluido en el árbol generador minimal.

**Optimalidad del algoritmo de Kruskal**

**Demostración**

Caso base: Sea $k_1$ la arista de menor peso en A. Entonces, existe un AGM tal que $k_1 \in T$.

Suponemos que es cierto para n-1: La (n-1)-ésima arista incluida por el algoritmo de Kruskal pertenece al AGM.

Demostramos que es cierto para n: La n-ésima arista incluida por el algoritmo de Kruskal pertenece al AGM.





**Optimalidad del algoritmo de Kruskal**

**Demostración**

Caso base

Por reducción al absurdo:

Sea $k_1$ la arista de menor peso en A.

Supongamos un AGM T′ que no incluye a $k_1$.

Consideremos $T' \cup k_1$ con $peso(T' \cup k_1) = peso(T') + peso(k_1)$.

En $T' \cup k_1$ aparece un ciclo (¿por qué?), pero si eliminamos cualquier arista del ciclo (x), distinta de $k_1$, obtenemos un árbol $T^* = T' + k_1 - x$ con $peso(T^*) = peso(T') + peso(k_1) – peso(x)$.

Por tanto, como $peso(k_1) < peso(x)$, deducimos que $peso(T^*) < peso(T')$. Contradicción.

# Algoritmo de Kruskal

## Optimalidad del algoritmo de Kruskal

## Demostración

Inducción

Por reducción al absurdo:

Supongamos un AGM T' que incluye a $\{k_1,..., k_{n-1}\}$ pero no incluye a $k_n$.

Consideremos $T' \cup k_n$ con $peso(T' \cup k_n) = peso(T') + peso(k_n)$.

Aparece un ciclo, que incluirá al menos una arista x que **NO** pertenece al conjunto de aristas seleccionadas $\{k_1, ..., k_{n-1}\}$

Eliminando dicha arista del ciclo, obtenemos un árbol $T^*=T'+k_n-x$ con $peso(T^*) = peso(T') + peso(k_n) – peso(x)$.

Pero $peso(k_n) < peso(x)$, por lo que $peso(T^*) < peso(T')$.
Contradicción.



Árboles generadores minimales

# Algoritmo de Kruskal

```
función Kruskal( Grafo G(V,A) )                         // Eficiencia
{
  set<aristas> C(A);
  set<aristas> S;

  Ordenar(C);                                            // O(A log A)

  while (!C.empty() && S.size()!=V.size()-1) {           // O(1)
        x = C.first();                                   // O(1)
        C.erase(x);                                      // O(1)
        if (!HayCiclo(S,x))                              // O(V)
            S.insert(x);                                 // O(V)
  }

  if (S.size()==V.size()-1)
     return S;
  else
     return "No_hay_solucion";
}
```

O(AV)

# Algoritmo de Kruskal

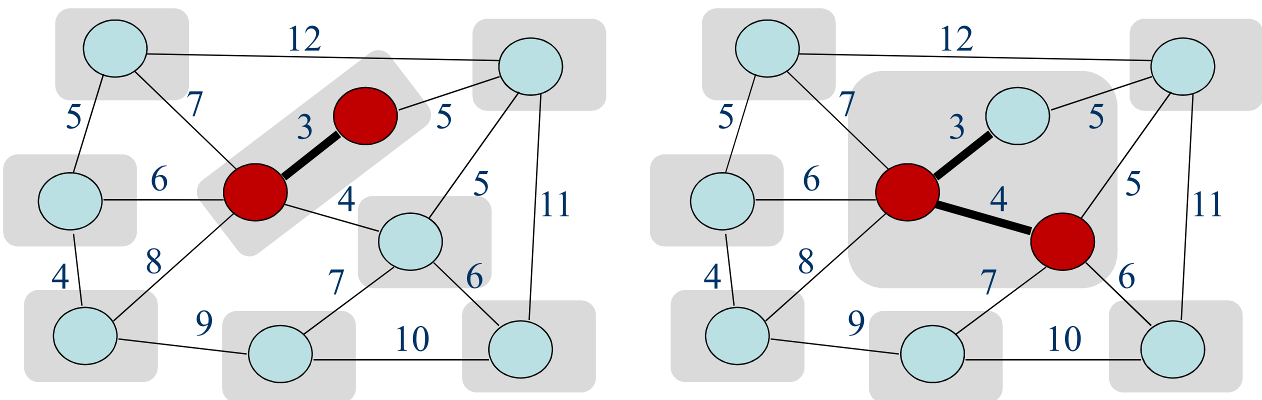
**Implementación eficiente del algoritmo de Kruskal (como combinación de componentes conexas)**

- Se comienza con un conjunto de n componentes conexas de tamaño 1 (cada nodo en una componente conexa).
- La función de factibilidad me aceptará la arista de menor costo que una dos componentes conexas (para garantizar que no hay ciclos).
- En cada iteración quedará una componente conexa menos, por lo que, finalmente, el algoritmo terminará con una única componente conexa: el árbol generador minimal.

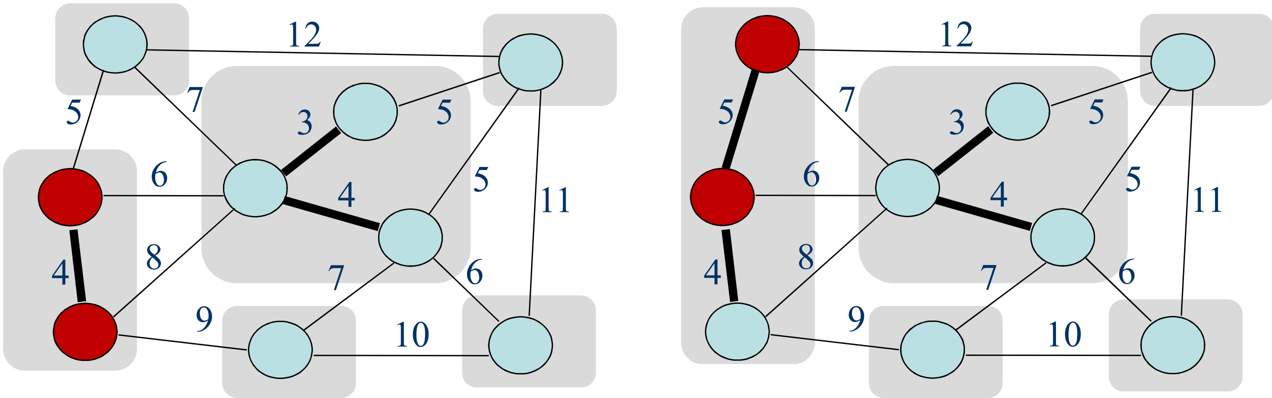




Algoritmo de Kruskal

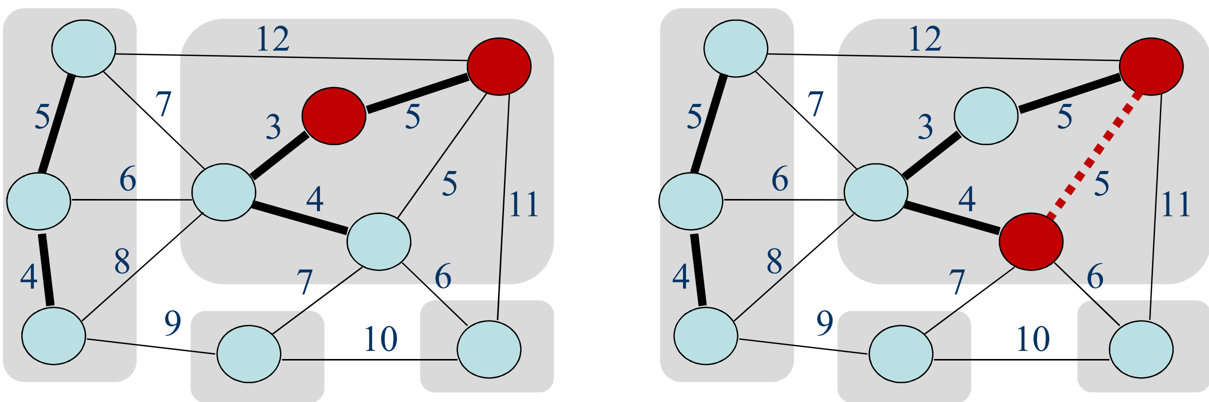


# Árboles generadores minimales
## Algoritmo de Kruskal

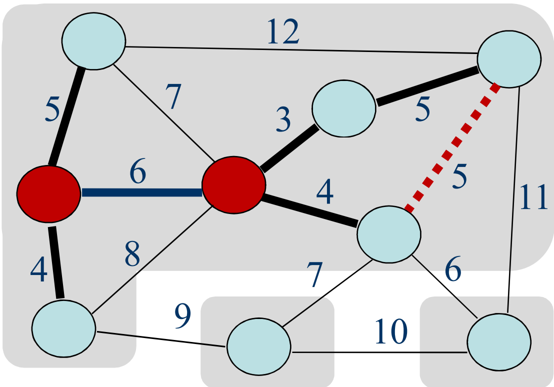


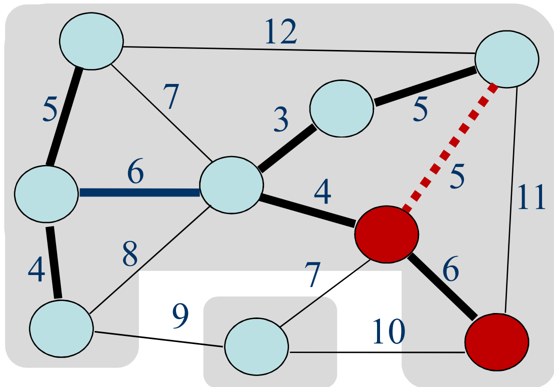



# Árboles generadores minimales
## Algoritmo de Kruskal



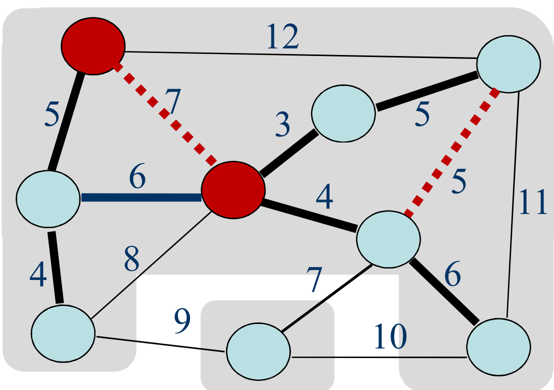
# Árboles generadores minimales
## Algoritmo de Kruskal

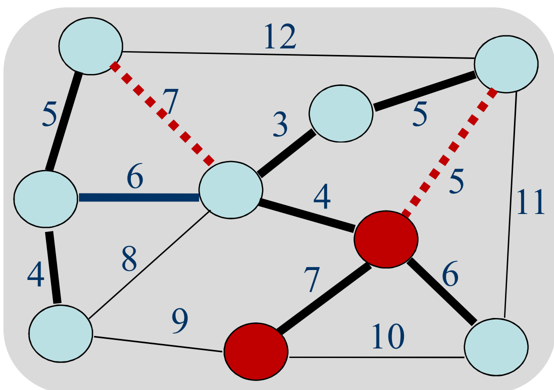





# Árboles generadores minimales
## Algoritmo de Kruskal



# Árboles generadores minimales
## Algoritmo de Kruskal

```
función Kruskal( Grafo G(V,A) )
{
  S = ∅;
  Ordenar(A);  // Orden creciente de pesos          // O(A log A)

  for (i=0; i<V.size()-1; i++)
     MakeSet(V[i]);                                  // O(1)
  while (!A.empty() && S.size()!=V.size()-1) {
        (u,v) = A.first();                           // O(1)
        if (FindSet(u) != FindSet(v))                // O(1)
           S = S ∪ {(u,v)};                          // O(1)
           Union(u,v);                               // O(V) ???
        }
  }
  if (S.size()==V.size()-1)
     return S;
  else
     return "No hay solución";
}
```

Cuello de botella del algoritmo

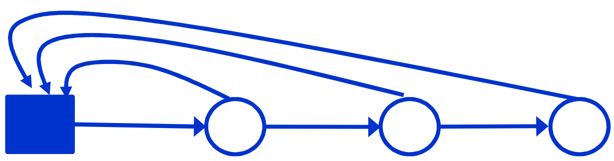


# Árboles generadores minimales
## Algoritmo de Kruskal

**Representación de conjuntos disjuntos**

Estructura de datos "union-find":

Listas enlazadas de elementos con punteros hacia el conjunto al que pertenecen



- MakeSet(): Creación del conjunto, O(1).
- FindSet(): Encontrar el conjunto al que pertenece, O(1).
- Union(A,B): "Copia" elementos de A a B haciendo que los elementos de A también apunten a B...

# Árboles generadores minimales
# Algoritmo de Kruskal

**Representación de conjuntos disjuntos**

Estructura de datos "union-find": Union(A,B)

¿Cuánto tardan en realizarse las n uniones?

- Análisis del peor caso: $O(n^2)$
  - Union($S_1$, $S_2$) "copia" 1 elemento.
  - Union($S_2$, $S_3$) "copia" 2 elementos.
  - Union($S_{n-1}$, $S_n$) "copia" n-1 elementos.
- Mejora: Copiar siempre el menor en el mayor.
  - Peor caso: Un elemento se copia como máximo log(n) veces, luego n uniones se hacen en O(n log n).
  - El análisis amortizado de la operación nos dice que una unión es de orden **O(log n)**.



# Árboles generadores minimales
# Algoritmo de Kruskal

```
función Kruskal( Grafo G(V,A) )
{
  S = ∅;
  Ordenar(A);  // Orden creciente de pesos          // O(A log A)

  for (i=0; i<V.size()-1; i++)
      MakeSet(V[i]);                                // O(1)
  while (!A.empty() && S.size()!=V.size()-1) {
        (u,v) = A.first();                          // O(1)
        if (FindSet(u) != FindSet(v))               // O(1)
           S = S ∪ {(u,v)};                         // O(1)
           Union(u,v);                              // O(log V)
        }
  }
  if (S.size()==V.size()-1)
     return S;
  else
     return "No hay solución";
}
```

O(A log V)

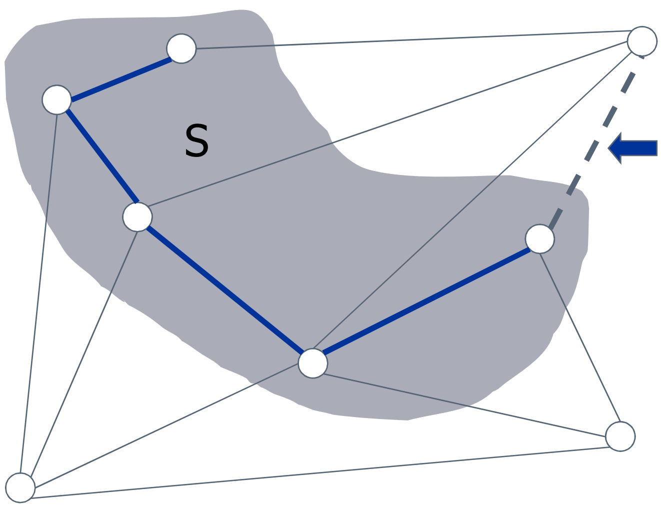
# Árboles generadores minimales
## Algoritmo de Prim

**Elementos del algoritmo de Prim**

- Conjunto de candidatos: Vértices del grafo.
- Función de selección: El vértice u aún no seleccionado que se conecte mediante la arista de menor peso a un vértice v del conjunto de vértices seleccionados.
- Función de factibilidad:
  El conjunto de aristas no contiene ningún ciclo.
- Criterio que define lo que es una solución: n-1 aristas.
  El conjunto de aristas (u,v) conecta todos los vértices.
- Función objetivo: Suma de los costes de las aristas.

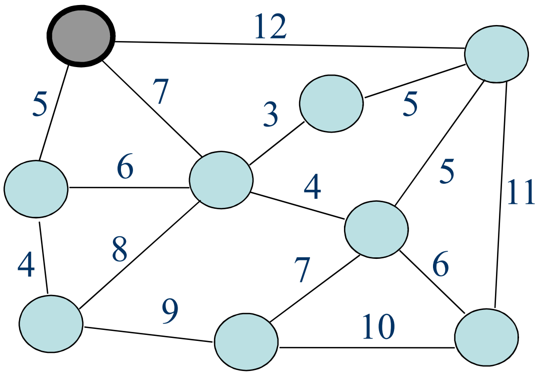


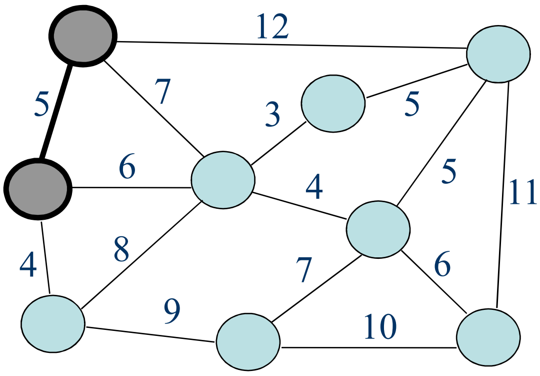
# Árboles generadores minimales
## Algoritmo de Prim




En cada iteración, añadimos la arista de menor coste que añade un nuevo nodo a nuestro árbol S…

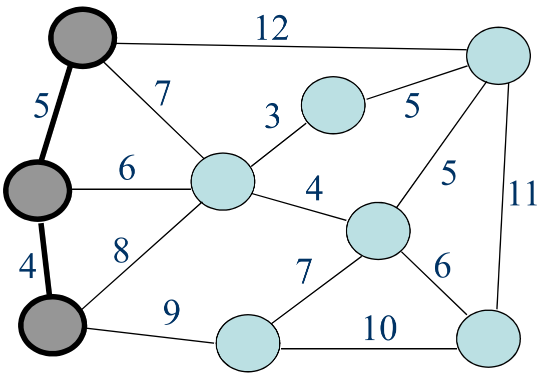
# Árboles generadores minimales
## Algoritmo de Prim

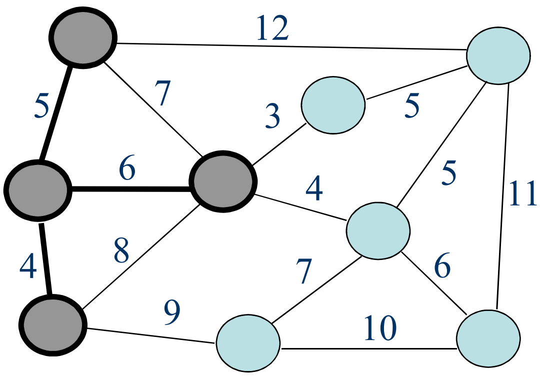


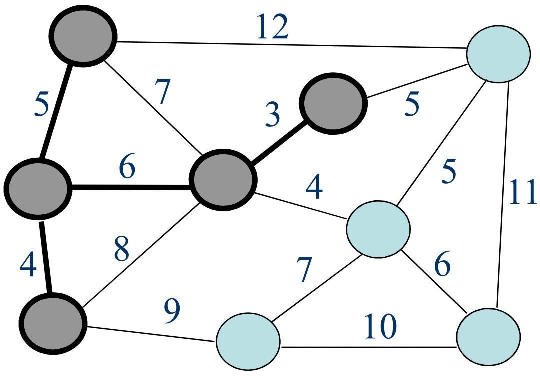


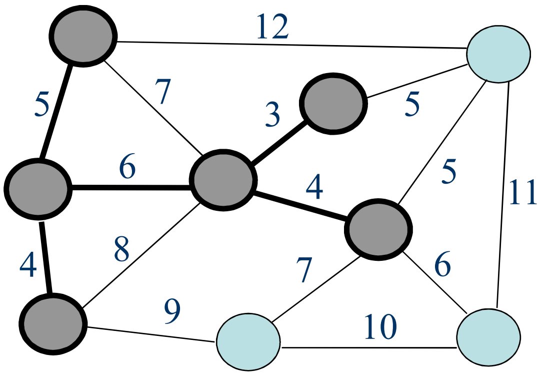

# Árboles generadores minimales
## Algoritmo de Prim



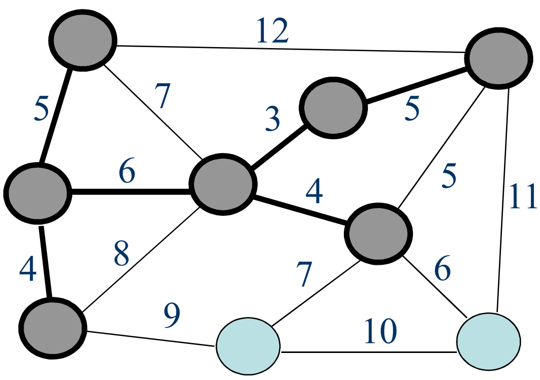
# Árboles generadores minimales
## Algoritmo de Prim

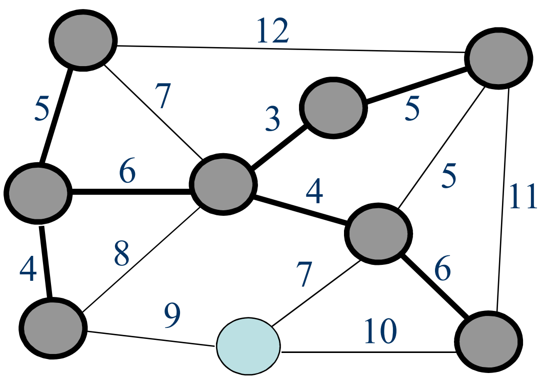


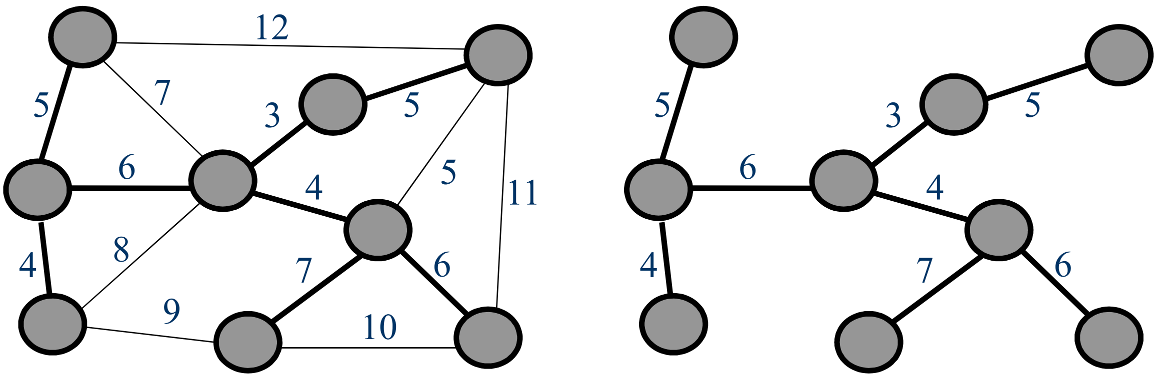






# Árboles generadores minimales
## Algoritmo de Prim










**Optimalidad del algoritmo de Prim**

Teorema:

Sean T un AGM de G=(V,A), S $\subseteq$ T un subárbol de T y (u,v) la arista de menor peso conectando los vértices de S con los de V-A. Entonces, (u,v) $\in$ T.

Demostración:

El teorema anterior se puede demostrar fácilmente si tenemos en cuenta que un AGM tiene subestructuras optimales…

# Árboles generadores minimales
# Algoritmo de Prim

**Optimalidad del algoritmo de Prim**

Teorema: Sea T un AGM y (u,v) una arista de T.
Si $T_1$ y $T_2$ son los dos árboles que se obtienen al eliminar la arista (u,v) de T, entonces $T_1$ es un AGM de $G_1=(V_1,A_1)$, y $T_2$ es un AGM de $G_2 = (V_2,A_2)$

Demostración:

Por reducción al absurdo: Si tenemos en cuenta que
peso(T) = peso(u,v) + peso($T_1$) + peso($T_2$),
no puede haber árboles generadores minimales mejores que $T_1$ o $T_2$, pues si los hubiese T no sería óptimo.





**Implementación del algoritmo de Prim**

Clave:

Seleccionar eficientemente la arista
que se añadirá al árbol generador minimal.

Solución:

Utilizar una cola con prioridad en la que tengamos los vértices asociados al menor coste de una arista que conecte cada vértice con un vértice que ya forme parte del AGM (infinito si no existiese dicha arista).

# Árboles generadores minimales
## Algoritmo de Prim

```
función Prim( Grafo G(V,A) )
{
  PriorityQueue Q;                    // Cola con prioridad
  foreach (v ∈ V) {
     coste[v] = ∞; padre[v] = NULL; Q.add(v, coste[v]);
  }
  coste[r] = 0;                       // Elección de una raíz r
  S = ∅;                              // Nodos ya explorados
  while (!Q.empty()) {
      u = Q.pop();                    // Menor elemento de Q
      S = S ∪ {u};
      foreach ((u,v)∈A incidente en u)
          if ((v∉S) && (coste(u,v) < coste[v])) {
             coste[v] = coste(u,v);   // Actualizar "prioridad"
             padre[v] = u;            // Vecino más cercano de u
          }
  }
  Resultado: El AGM está almacenado en el vector de padres
}
```



# Árboles generadores minimales
## Algoritmo de Prim

```
función Prim( Grafo G(V,A) )
{
  PriorityQueue Q;
  foreach (v ∈ V) {                   // O(V log V)
     coste[v] = ∞; padre[v] = NULL; Q.add(v, coste[v]);
  }
  coste[r] = 0;                       // O(log V)
  S = ∅;
  while (!Q.empty()) {                // V iteraciones
      u = Q.pop();                    // O(log V)
      S = S ∪ {u};
      foreach ((u,v)∈A incidente en u)
          if ((v∉S) && (coste(u,v) < coste[v])) {
             coste[v] = coste(u,v);   // O(log V)
             padre[v] = u;            // O(1)
          }
  }
  Resultado: El AGM está almacenado en el vector de padres
}
```

Para cada arista del grafo

O(A log V)

# Algoritmo de Prim

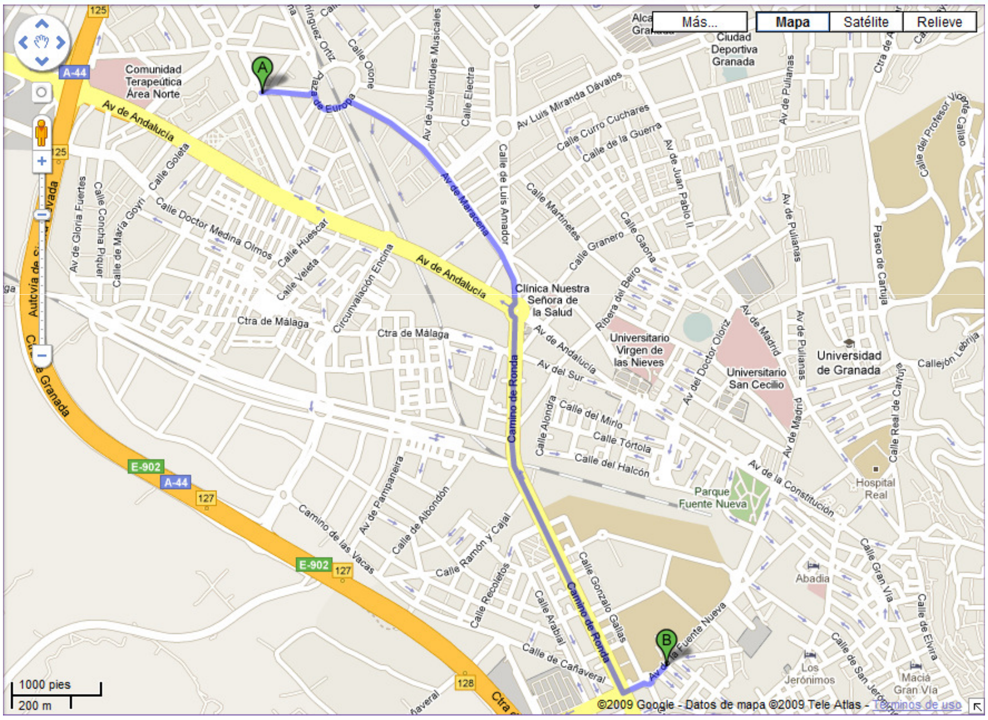
**Eficiencia del algoritmo de Prim**

O(A log V + V log V) = O(A log V)

ya que, en un grafo conexo, V-1 ≤ A ≤ V(V-1)



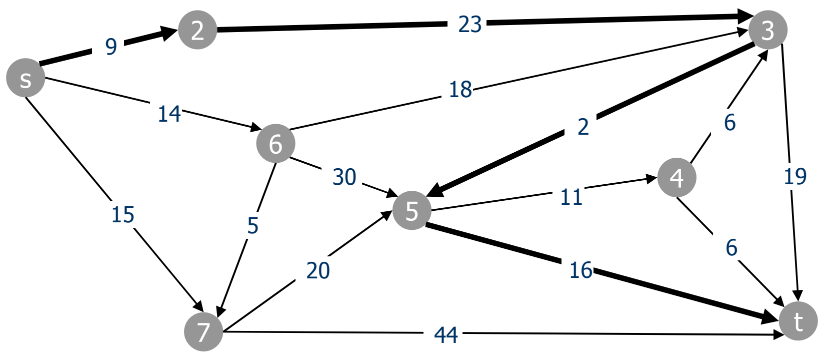
# Caminos mínimos




De la ETSIIT a la Facultad de Ciencias… 2.9 km

# Caminos mínimos

**Problema**
Dado un grafo G ponderado con pesos positivos, calcular el camino de menor peso existente entre un vértice s y otro vértice t.




Camino más corto
s-2-3-5-t

Coste = 50
9 + 23 + 2 + 16



# Caminos mínimos

**Algoritmo de Dijkstra** (1959)

Dado un grafo G=(V,A) y un vértice s,
encontrar el camino de costo mínimo
para llegar desde s al resto de los vértices en el grafo.

IDEA:

Mantener el conjunto de nodos ya explorados para los cuales ya hemos determinado el camino más corto desde s…

# Caminos mínimos

**Algoritmo de Dijkstra** (1959)

- Conjunto de candidatos: Vértices del grafo.
- Solución parcial S: Vértices a los cuales ya sabemos llegar usando el camino más corto (inicialmente ∅)
- Función de selección: Vértice v del conjunto de candidatos (V\S) que esté más cerca del vértice s.

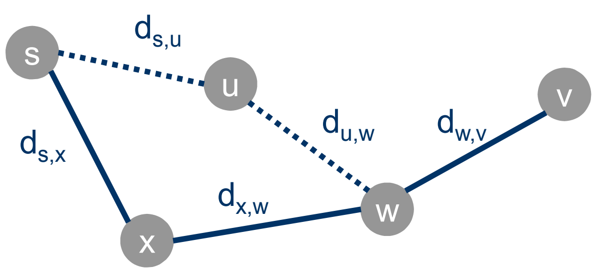


# Caminos mínimos

**Propiedades de los caminos mínimos**

Si d(s,v) es la longitud del camino mínimo para ir desde el vértice s hasta el vértice v, entonces se satisface que

$$d(s,v) \leq d(s,u) + d(u,v)$$



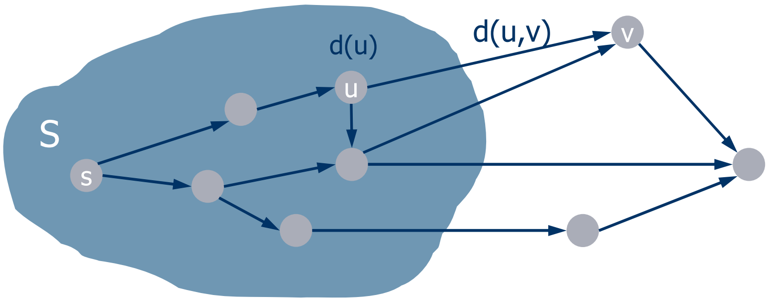
# Caminos mínimos

**Algoritmo de Dijkstra** (1959)

- Función de selección: Vértice v del conjunto de candidatos (V\S) que esté más cerca del vértice s.

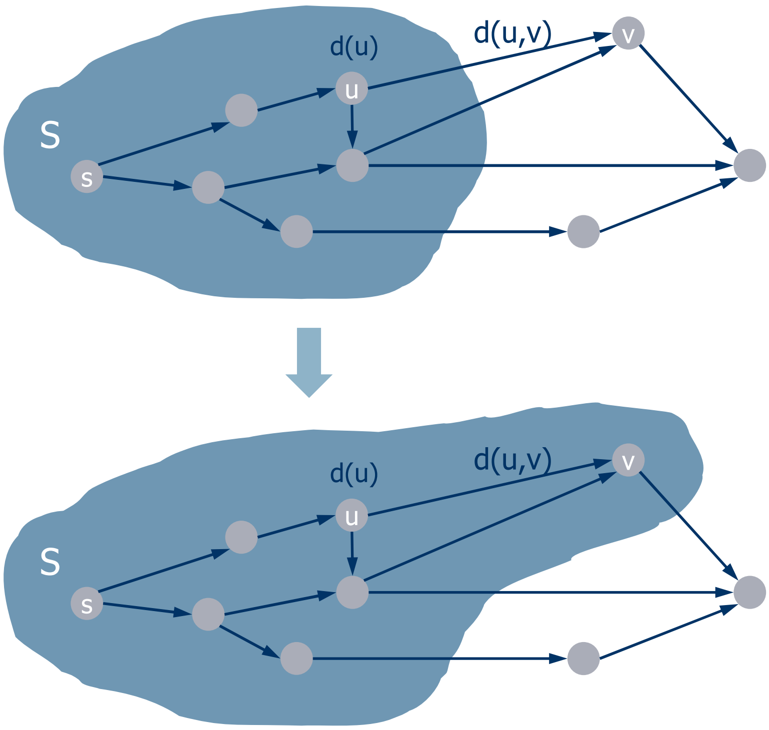
  Esto es, elegir el vértice v que minimice

$$\pi(v) = \min_{(u,v):u\in S} d(u) + d(u,v)$$






# Caminos mínimos



# Caminos mínimos
## Algoritmo de Dijkstra

```
función Dijkstra ( Grafo G(V,A), vértice s )
{
  Set           S = ∅;        // Vértices ya seleccionados
  PriorityQueue Q;            // Cola con prioridad
  foreach ( v∈V ) {
          d[v] = ∞; pred[v] = null; Q.add(v,d[v]);
  }
  d[s]=0;
  while (!Q.empty()) {
        v = Q.pop ();         // Selección del vértice
        S.add(v);
        foreach ( (v,w)∈A incidente en v )
                if ( d[w] > d[v] + coste(v,w) ) {
                   d[w] =  d[v] + coste(v,w);
                   pred[w] = v;
                }
  }
  Resultado: Caminos mínimos almacenados en el vector pred[]
}
```



# Caminos mínimos
## Algoritmo de Dijkstra

```
función Dijkstra ( Grafo G(V,A), vértice s )
{
  Set           S = ∅;
  PriorityQueue Q;
  foreach ( v∈V ) {
          d[v] = ∞; pred[v] = null; Q.add(v,d[v]);   // O(log V)
  }
  d[s]=0;                                             // O(1)
  while (!Q.empty()) {
        v = Q.pop ();                                 // O(log V)
        S.add(v);                                     // O(1)
        foreach ( (v,w)∈A incidente en v )
                if ( d[w] > d[v] + coste(v,w) ) {     // O(1)
                   d[w] =  d[v] + coste(v,w);         // O(log V)
                   pred[w] = v;                       // O(1)
                }
  }
  Resultado: Caminos mínimos almacenados en el vector pred[]
}
```

# Caminos mínimos
## Algoritmo de Dijkstra

```
función Dijkstra ( Grafo G(V,A), vértice s )
{
  Set           S = ∅;
  PriorityQueue Q;
  foreach ( v∈V ) {                              // O(V log V)
          d[v] = ∞; pred[v] = null; Q.add(v,d[v]);
  }
  d[s]=0;
  while (!Q.empty()) {
        v = Q.pop ();
        S.add(v);
        foreach ( (v,w)∈A incidente en v )
                if ( d[w] > d[v] + coste(v,w) ) {
                   d[w] =  d[v] + coste(v,w);
                   pred[w] = v;
                }
  }
  Resultado: Caminos mínimos almacenados en el vector pred[]
}
```

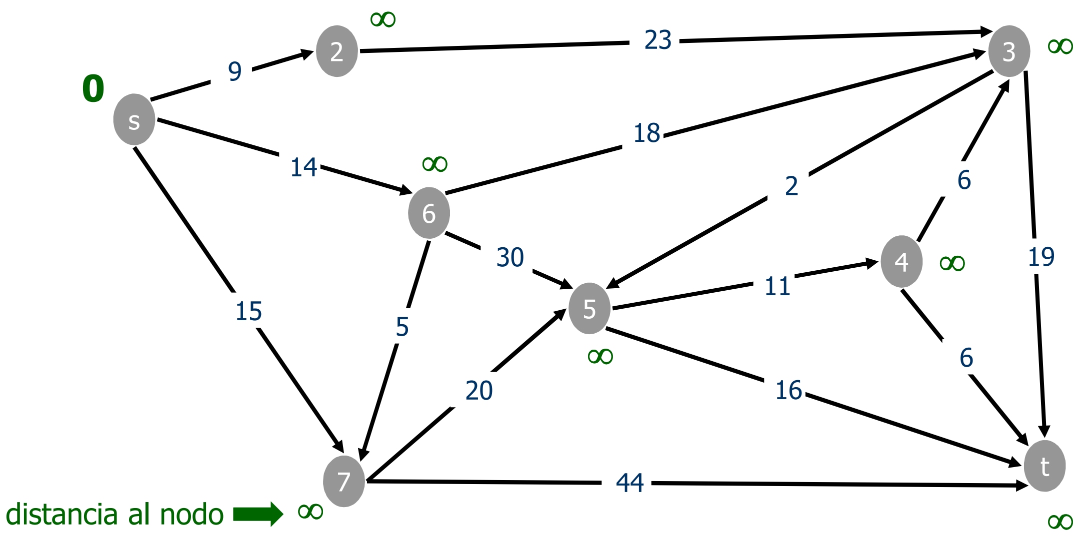
Para cada arista del grafo
O(A log V)



# Caminos mínimos

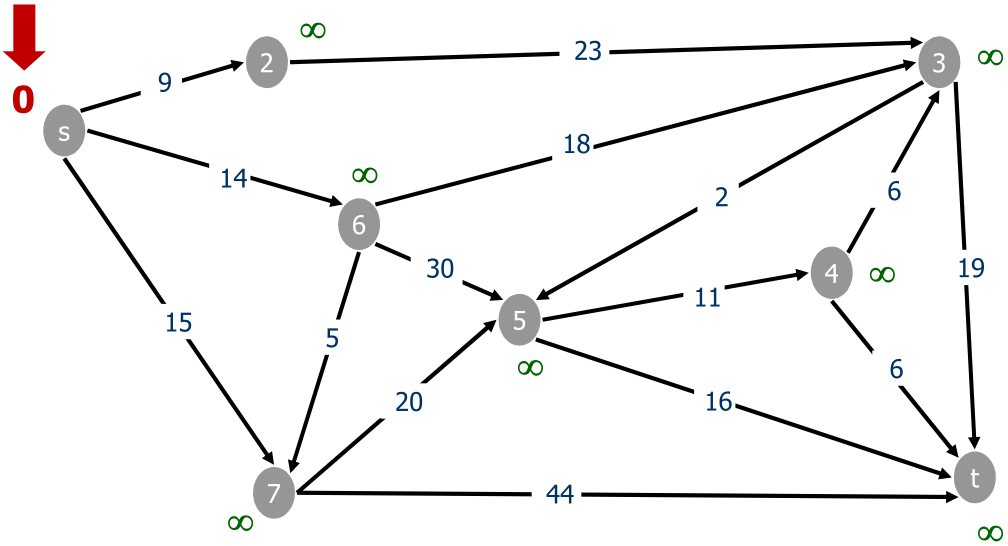
S = ∅

Q = {s, 2, 3, 4, 5, 6, 7, t }



# Caminos mínimos

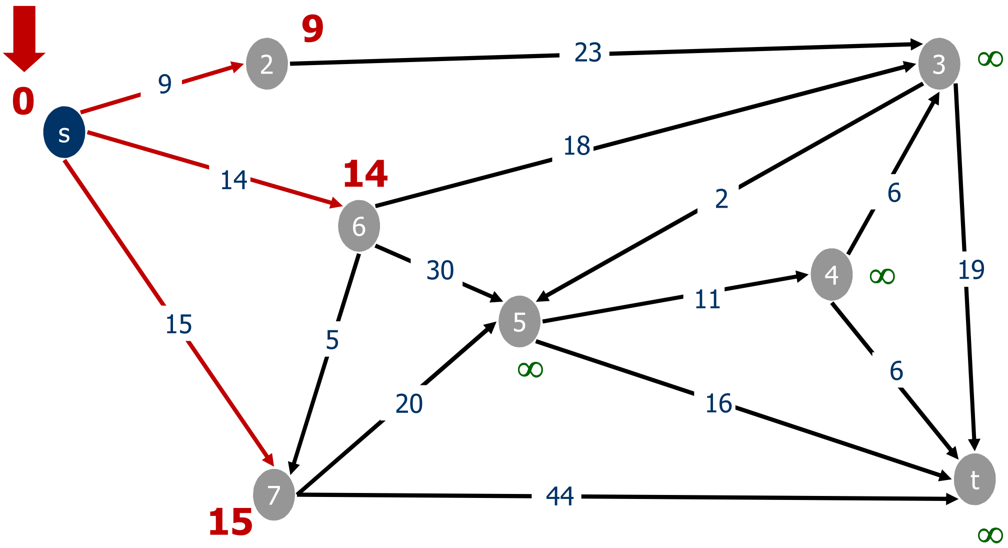
S = { }
Q = { s, 2, 3, 4, 5, 6, 7, t }






# Caminos mínimos

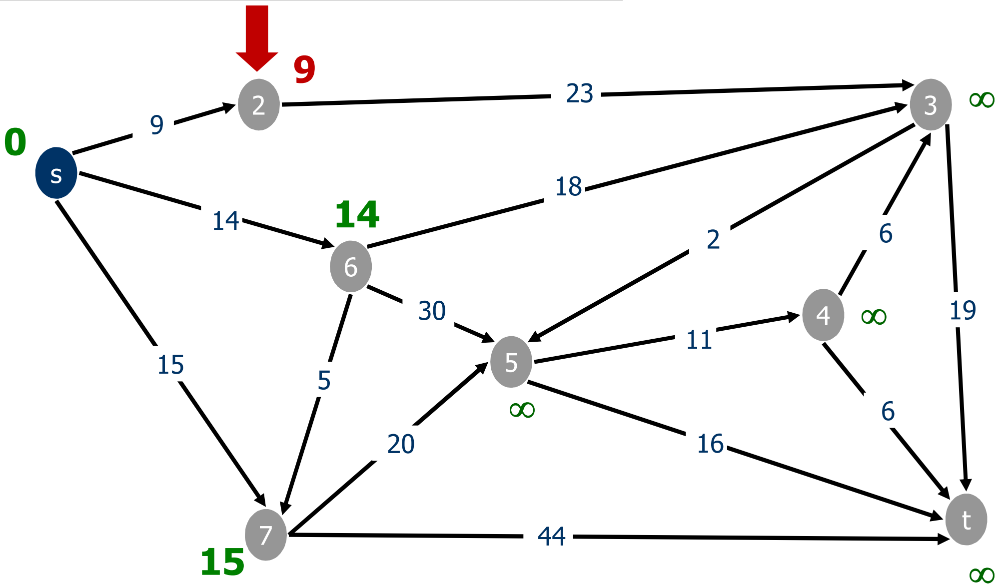
S = { s }
Q = { 2, 3, 4, 5, 6, 7, t }



# Caminos mínimos

S = { s }

Q = { 2, 3, 4, 5, 6, 7, t }

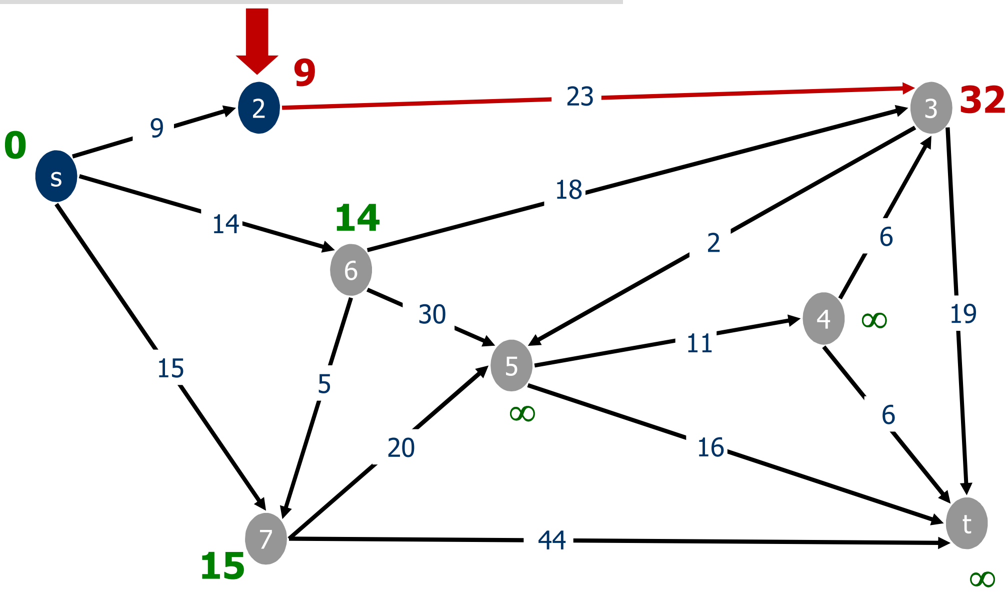





# Caminos mínimos

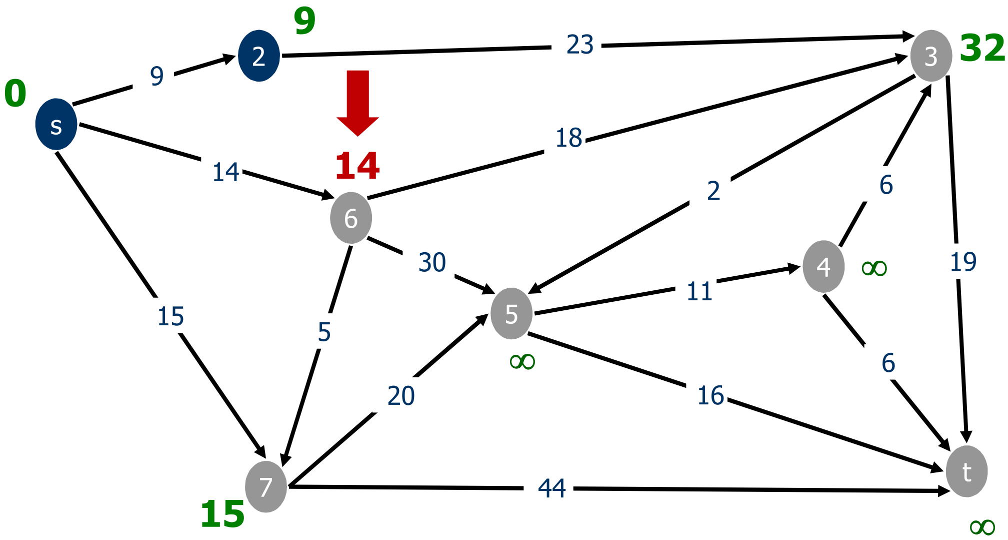
S = { s, 2 }

Q = { 3, 4, 5, 6, 7, t }



# Caminos mínimos

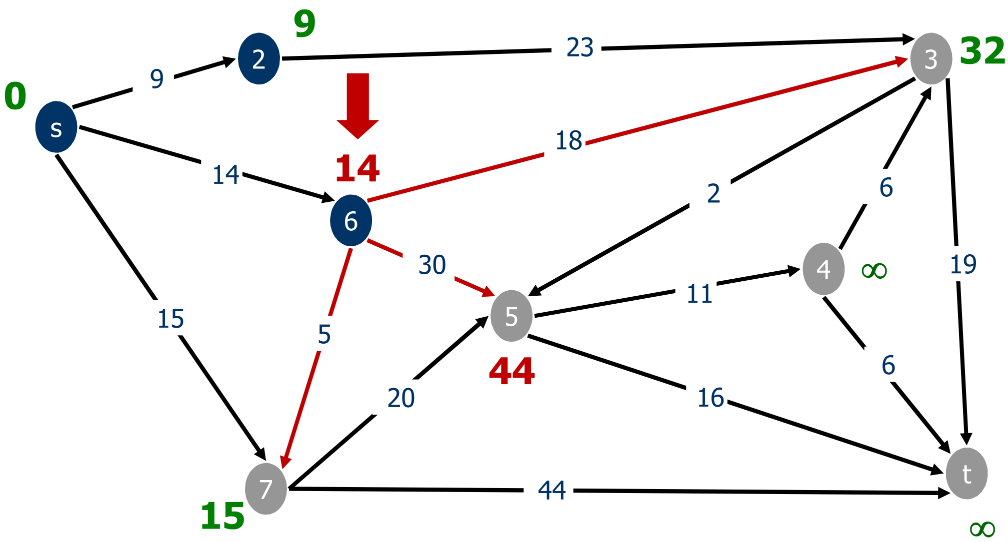
S = { s, 2 }
Q = { 3, 4, 5, 6, 7, t }





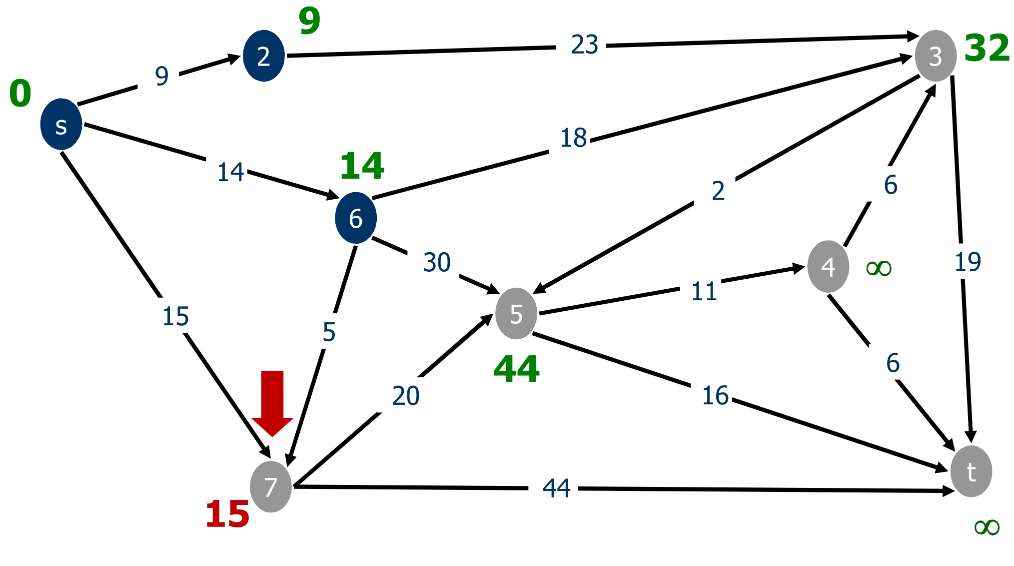

# Caminos mínimos

S = { s, 2, 6 }
Q = { 3, 4, 5, 7, t }



# Caminos mínimos

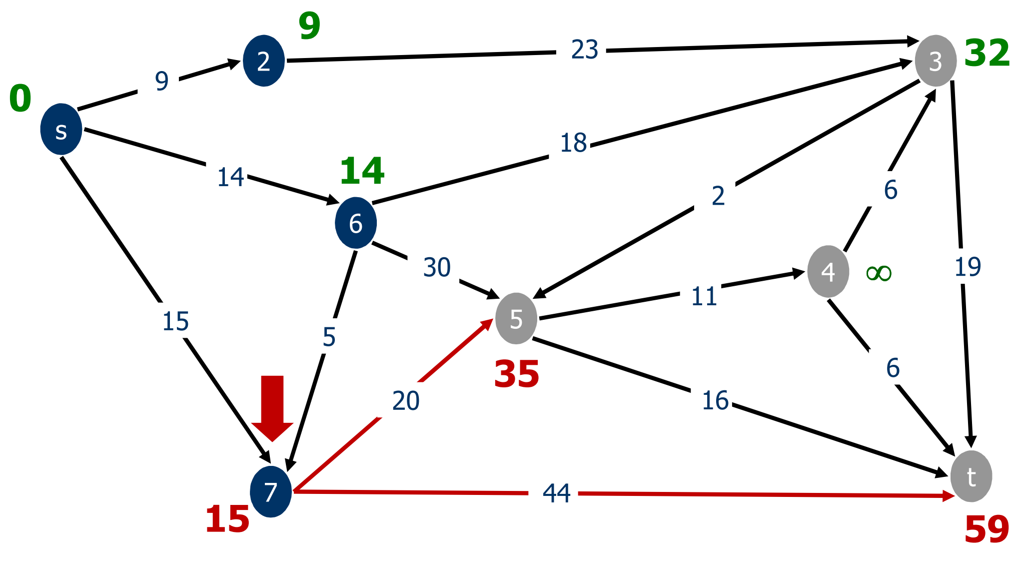
S = { s, 2, 6 }
Q = { 3, 4, 5, 7, t }





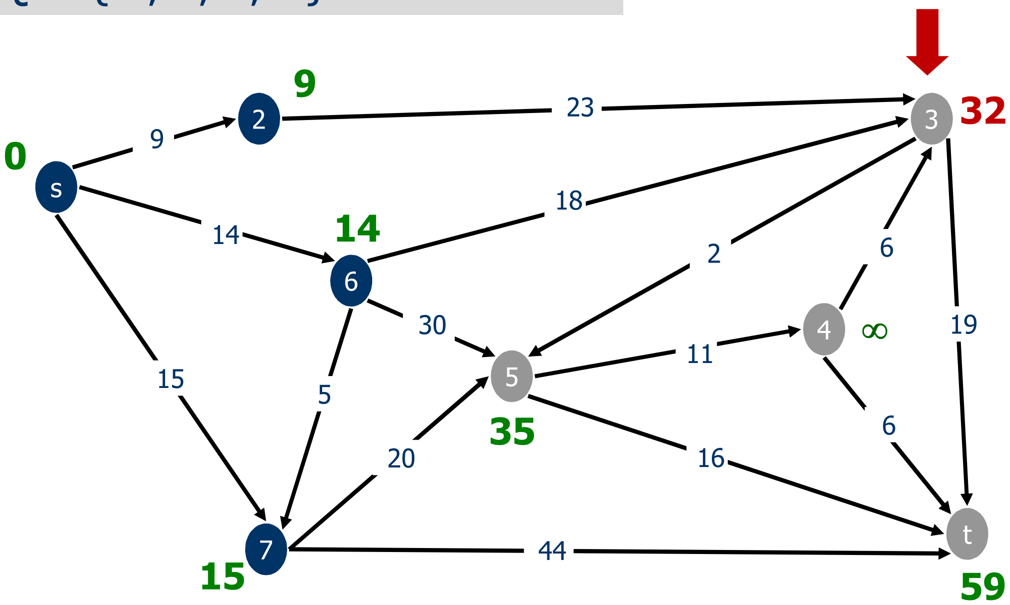

# Caminos mínimos

S = { s, 2, 6, 7 }
Q = { 3, 4, 5, t }



# Caminos mínimos

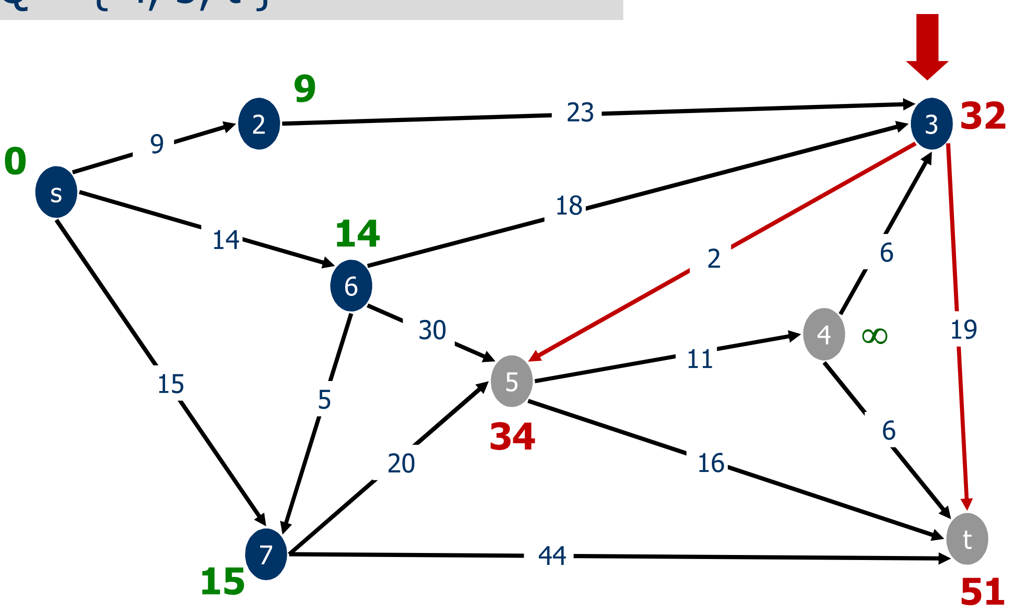
S = { s, 2, 6, 7 }
Q = { 3, 4, 5, t }





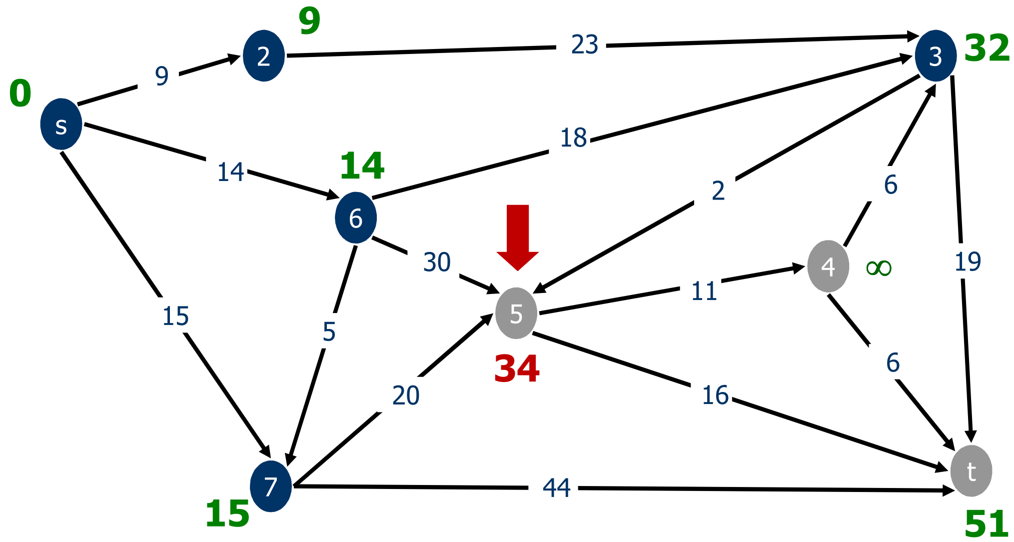

# Caminos mínimos

S = { s, 2, 3, 6, 7 }
Q = { 4, 5, t }



# Caminos mínimos

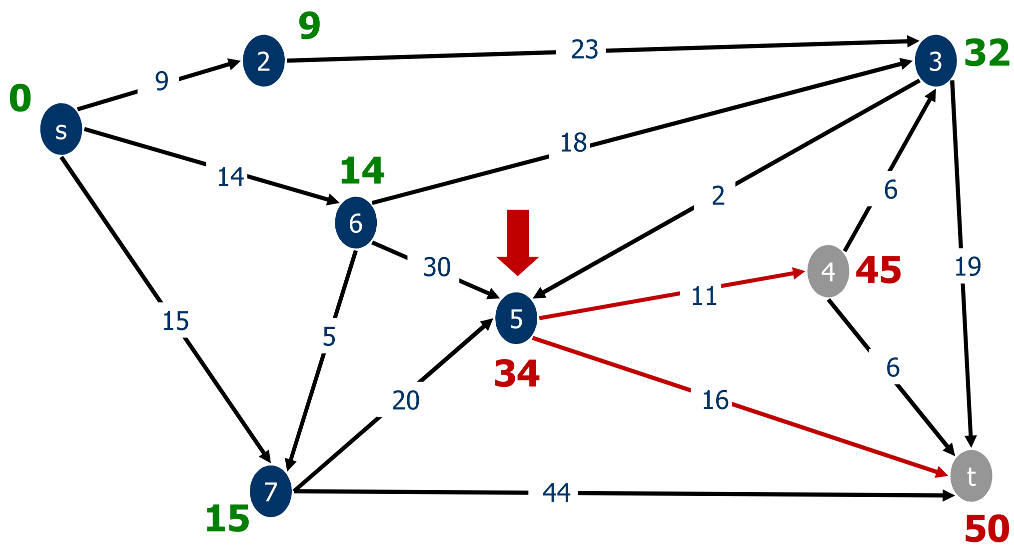
S = { s, 2, 3, 6, 7 }
Q = { 4, 5, t }





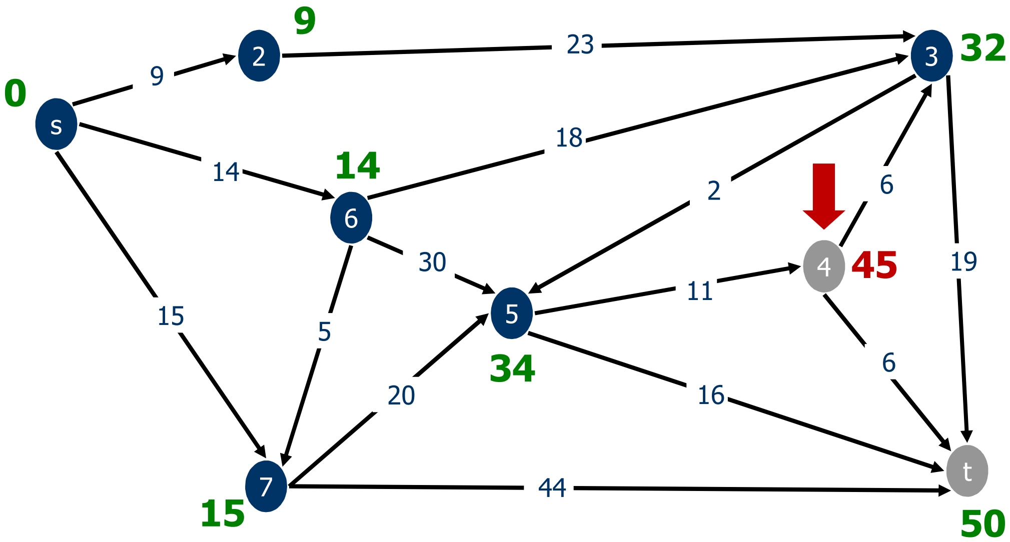

# Caminos mínimos

S = { s, 2, 3, 5, 6, 7 }
Q = { 4, t }



# Caminos mínimos

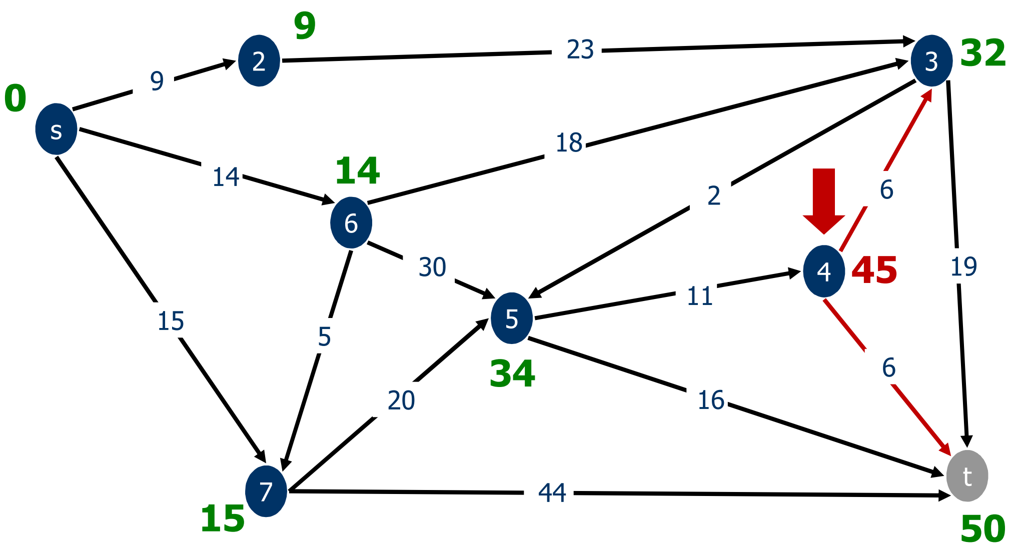
S = { s, 2, 3, 5, 6, 7 }
Q = { 4, t }





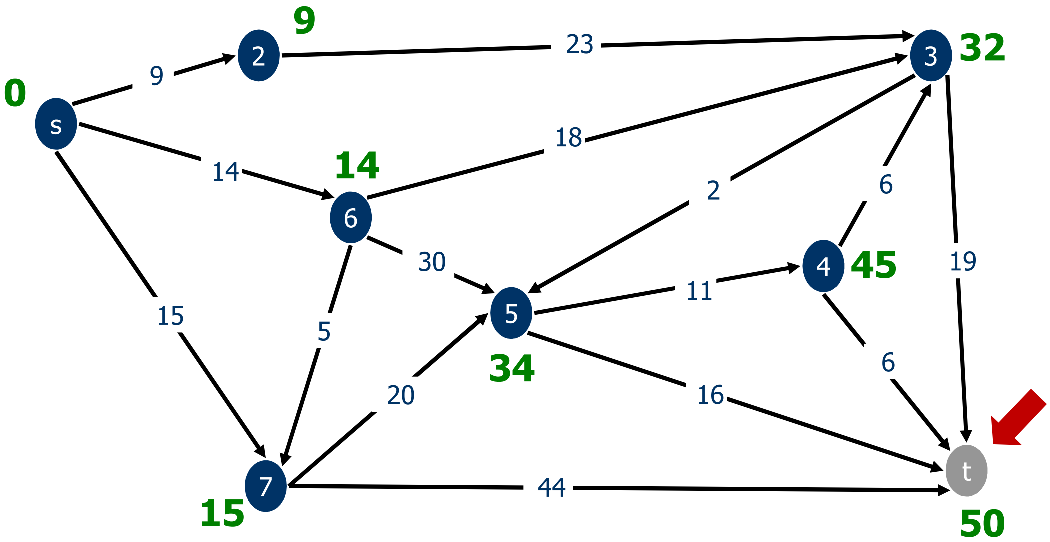

# Caminos mínimos

S = { s, 2, 3, 4, 5, 6, 7 }
Q = { t }



# Caminos mínimos

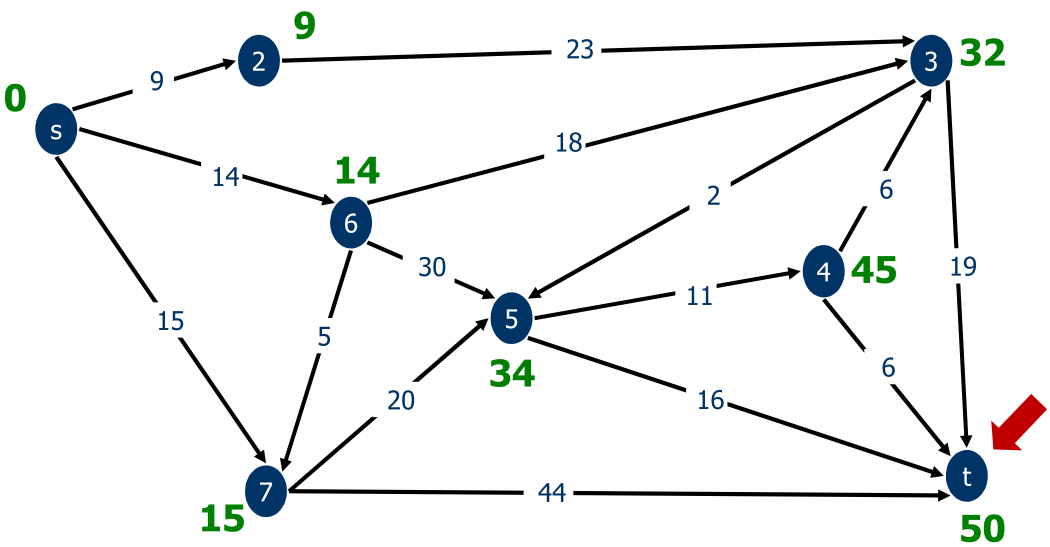
S = { s, 2, 3, 4, 5, 6, 7 }
Q = { t }



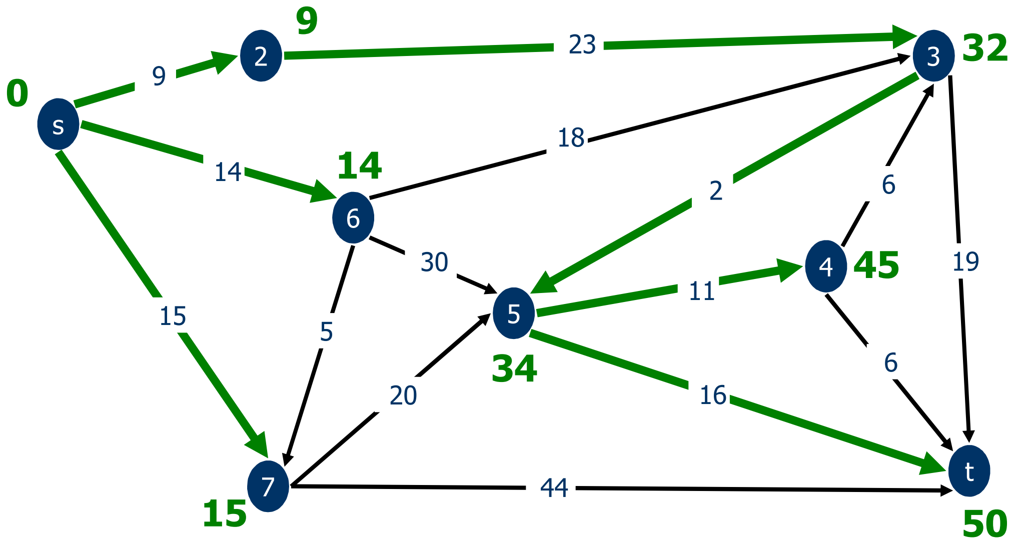



# Caminos mínimos

S = { s, 2, 3, 4, 5, 6, 7, t }
Q = { }



# Caminos mínimos



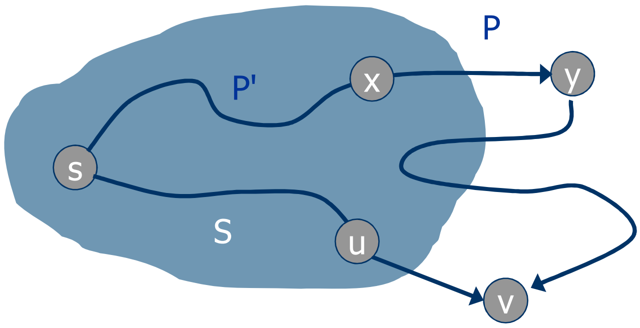



# Caminos mínimos

**Optimalidad del algoritmo de Dijkstra**

Invariante:

Para cada $v \in S$, $d(v)$ es la longitud del camino mínimo para ir desde el vértice s hasta el vértice v.



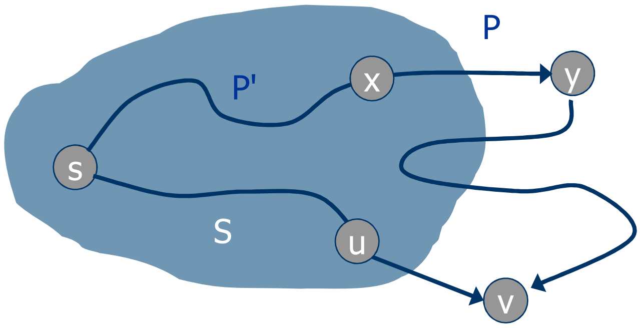
# Caminos mínimos

**Optimalidad del algoritmo de Dijkstra**

**Demostración**

Por inducción sobre el tamaño de S

Caso base: |S| = 0. Trivial.






# Caminos mínimos

**Optimalidad del algoritmo de Dijkstra**

**Demostración**

Por inducción sobre el tamaño de S

Inducción: Supongamos que es cierto para $|S| = k \geq 0$.

- Sea v el siguiente nodo que se añade a S y (u,v) la arista elegida para conectar S con v.
- El camino más corto de s a u más (u,v) es un camino de s a v de longitud d(v).
- Cualquier otro camino P de s a v será de longitud $\ell(P) \geq d(v)$ : Sea (x,y) la primera arista de P que abandona S y P' el subcamino de s a x. La longitud de P es mayor que d(v) en cuanto abandona S.

# Caminos mínimos

## Optimalidad del algoritmo de Dijkstra

## Demostración

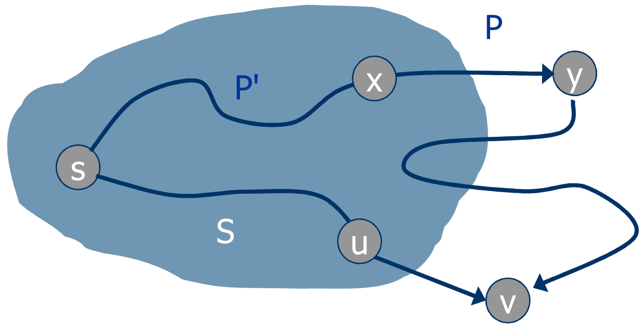
$$\ell(P) \ge \ell(P') + \ell(x,y) \ge d(x) + \ell(x,y) \ge d(y) \ge d(v)$$

Pesos no negativos

Hipótesis inductiva

Por definición

El algoritmo de Dijkstra elige v antes que y






# Caminos mínimos

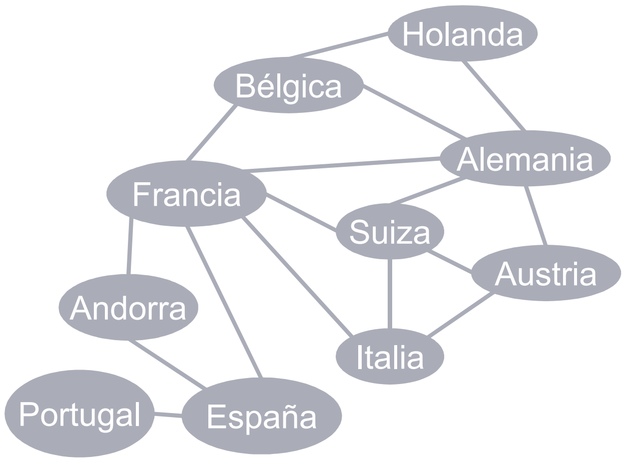
## Implementación del algoritmo de Dijkstra

La implementación eficiente del algoritmo de Dijkstra se basa en el uso de una estructura de datos adecuada: Una cola con prioridad.

| Operación | Algoritmo Dijkstra | Array | Heap binario | Heap n-ario | Heap Fibonacci† |
|---|---|---|---|---|---|
| add | V | V | log V | n $\log_n$ V | 1 |
| min | V | V | log V | n $\log_n$ V | log V |
| update | A | 1 | log V | $\log_n$ V | 1 |
| isEmpty | V | 1 | 1 | 1 | 1 |
| Total | | $V^2$ | A log V | A $\log_{A/D}$ V | A+V log V |

† Análisis amortizado

# Heurísticas greedy
## El problema del coloreo de un grafo




# Heurísticas greedy
## El problema del coloreo de un grafo

**Problema:**

Dado un grafo G=(V,A), se pretende colorear cada vértice de tal forma que dos vértices adyacentes no tengan el mismo color.

**Objetivo:**

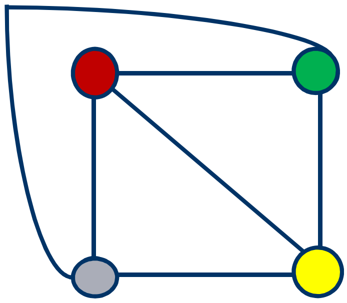
Minimizar el número de colores utilizados.

**Problema NP**

No existe ningún algoritmo eficiente que nos asegure haber utilizado un número mínimo de colores.

## El problema del coloreo de un grafo

**Teorema de Appel-Hanke** (1976): Un grafo plano requiere a lo sumo 4 colores para colorear sus nodos de forma que no haya vértices adyacentes del mismo color.



Si el grafo no es plano,
puede requerir tantos colores como vértices haya.





## El problema del coloreo de un grafo

Algoritmo greedy heurístico: **O(V)**

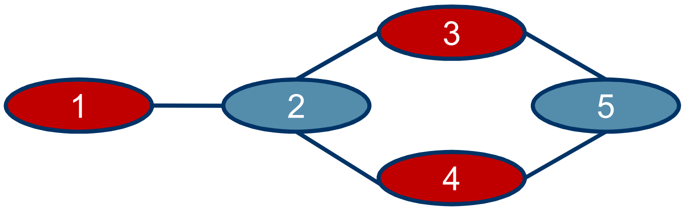
```
función Coloreo ( Grafo G(V,A) )
{
   i = 1;
   while (grafo no coloreado) {
          Elegir un color ci
          Colorear todos los vértices que se pueda con ci
           a partir de un vértice arbitrario (esto es, todos
           los vértices que no sean adyacentes a un vértice
           ya coloreado con ci)
          i = i + 1;
    }
}
```

## El problema del coloreo de un grafo

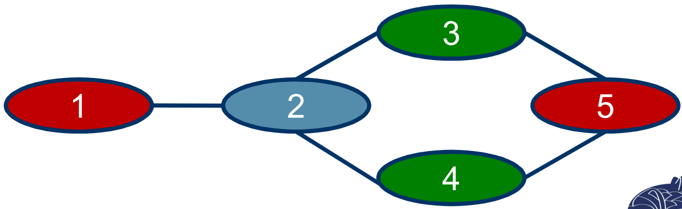
Algoritmo greedy heurístico:

El orden en que se escojan los nodos es decisivo...

**Solución óptima**




**Solución subóptima**



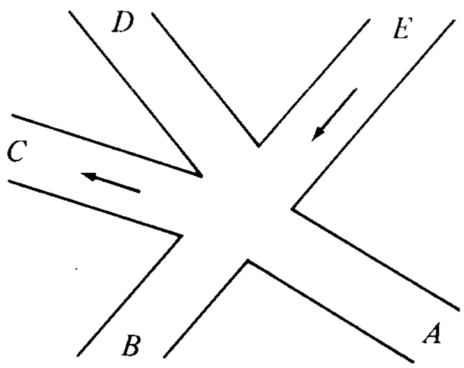





## El problema del coloreo de un grafo

Aplicación: Diseño de los semáforos de un cruce

En un cruce de calles señalamos los sentidos de circulación.




Para minimizar el tiempo de espera, construimos un grafo cuyos vértices representan turnos de circulación y cuyas aristas unen los turnos que no pueden realizarse simultáneamente sin que haya colisiones.

El problema del cruce con semáforos se convierte en un problema de coloreo de los vértices de un grafo.

**Problema:**

Dado un grafo G=(V,A), encontrar un camino que empiece en un vértice v y acabe en ese mismo vértice pasando una única vez por todos los vértices de V (es decir, un circuito hamiltoniano).

**Objetivo:**

Obtener el circuito hamiltoniano de coste mínimo.

**Problema NP**

No existe ningún algoritmo eficiente
para resolver el problema del viajante de comercio.





**Heurística greedy 1: Nodos como candidatos**

Escoger, en cada momento, el vértice más cercano al último nodo añadido al circuito (siempre que no se haya seleccionado previamente y que no cierre el camino antes de pasar por todos los vértices).

**Heurística greedy 2: Aristas como candidatas**

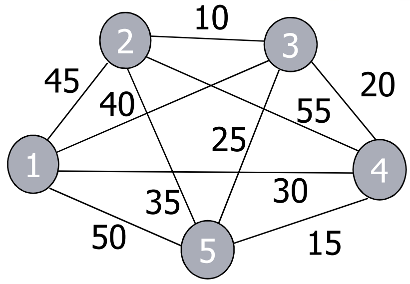
Como en el algoritmo de Kruskal, pero garantizando que al final se forme un circuito.

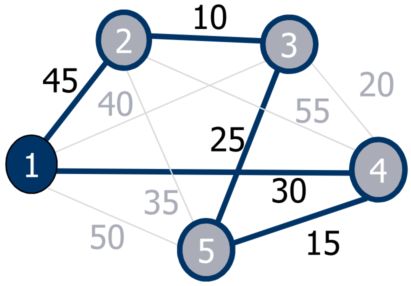
**Eficiencia:** La del algoritmo de ordenación que se use.

## El problema del viajante de comercio

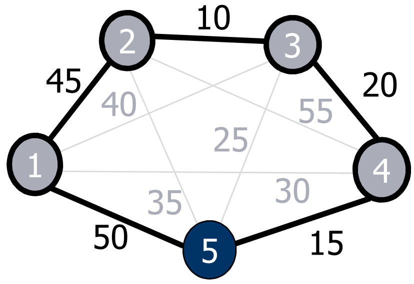
Primera heurística:

**Nodos como candidatos**




Empezando en el nodo 1
Circuito (1,4,5,3,2)
Solución = 125




Empezando en el nodo 5
Circuito (5,4,3,2,1)
Solución = 140

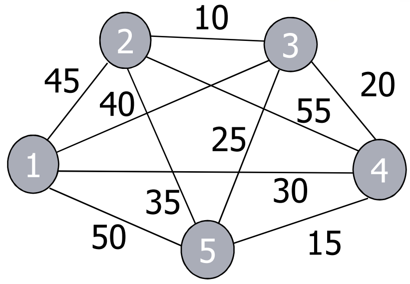


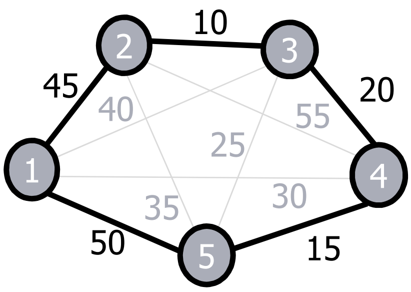


## El problema del viajante de comercio

Segunda heurística:

**Aristas como candidatas**




Circuito: (2,3), (4,5), (3,4), (1,2), (1,5)
Solución = 10 + 15 + 20 + 45 + 50 = 140