

# Algoritmos y Estructuras de Datos II

Algoritmos voraces

11 de mayo de 2016

## Clase de hoy

- 1 Algoritmos voraces
  - Árboles generadores de costo mínimo
  
- 2 El problema union-find
  - Primer intento
  - Segundo intento
  - Tercer intento
  - Último intento

## Algoritmos Voraces (o Glotones, Golosos) (Greedy)

- Es la técnica más sencilla de resolución de problemas.
- Normalmente se trata de algoritmos que resuelven problemas de **optimización**.
- Los algoritmos voraces intentan construir la solución óptima buscada **paso a paso**,
- **eligiendo** en cada paso
- la **componente** de la solución
- que **parece** más apropiada.

## Características

- Nunca revisan una **elección** ya realizada,
- confían en haber elegido bien las componentes anteriores.
- Por ello, lamentablemente, no siempre funcionan,
- pero cuando funcionan son muy eficientes.
- Existen varios problemas interesantes que admiten soluciones voraces,
- que como acabamos de decir, resultan muy eficientes.

## Problemas con solución voraz

- Problema de la moneda.
- Problema de la mochila.
- Problema del camino de costo mínimo en un grafo.
- Problema del árbol generador de costo mínimo en un grafo.

## Forma general

```
fun greedy(C) ret S
    {C: conjunto de candidatos, S: solución a construir}
    S:= {}
    do S no es solución → c:= seleccionar de C
        C:= C-{c}
        if S∪{c} es factible → S:= S∪{c} fi
    od
end fun
```

Lo más importante es el criterio de selección.

## Árbol generador de costo mínimo

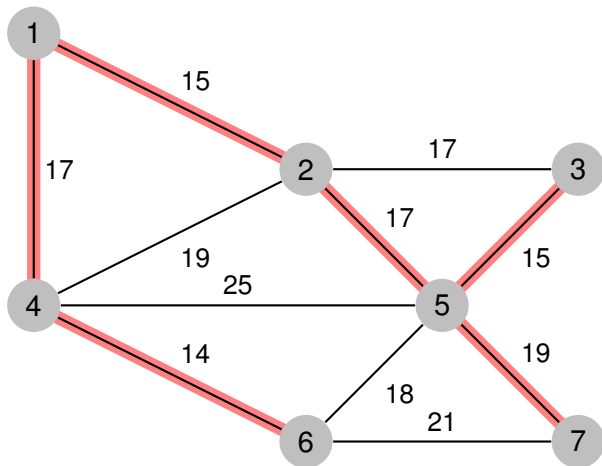
- Sea  $G = (V, A)$  un grafo conexo no dirigido con un costo no negativo asociado a cada arista.
- Se dice que  $T \subseteq A$  es un árbol generador (intuitivamente, un tendido) si el grafo  $(V, T)$  es conexo y no contiene ciclos.
- Su costo es la suma de los costos de sus aristas.
- Se busca  $T$  tal que su costo sea mínimo.

## Árbol generador de costo mínimo

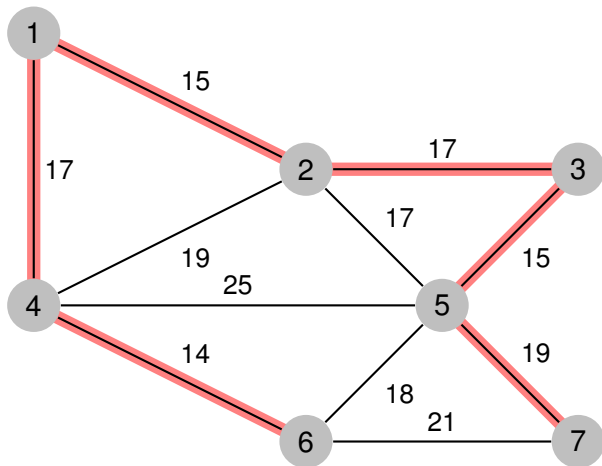
- El problema de encontrar un árbol generador de costo mínimo tiene numerosas aplicaciones en la vida real.
- Cada vez que se quiera realizar un tendido (eléctrico, telefónico, etc) se quieren unir distintas localidades de modo que requiera el menor costo en instalaciones (por ejemplo, cables) posible.
- Se trata de realizar el tendido siguiendo la traza de un árbol generador de costo mínimo.



# Ejemplo



# Ejemplo

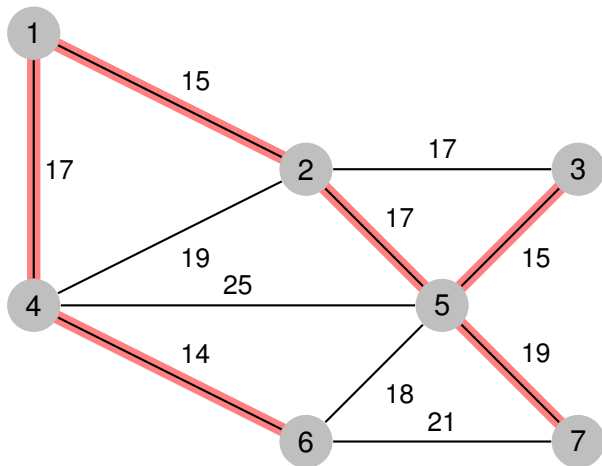


## Dos estrategias

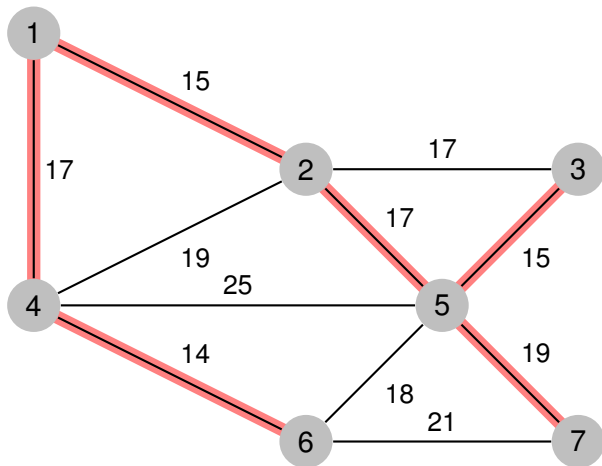
Hay dos grandes ideas de cómo resolverlo:

- La de Prim: se parte desde un vértice origen y se va extendiendo el tendido a partir de ahí:
  - en cada paso se une el tendido ya existente con alguno de los vértices aún no alcanzados, seleccionando la arista de menor costo capaz de incorporar un nuevo vértice
- La de Kruskal: se divide el grafo en distintas componentes (originariamente una por cada vértice) y se van uniendo componentes,
  - en cada paso se selecciona la arista de menor costo capaz de unir componentes.

# Algoritmo de Prim



# Algoritmo de Kruskal



# Implementación del Algoritmo de Prim

```
fun Prim( $G=(V,A)$  con costos en las aristas,  $k: V$ )  
    ret  $T$ : conjunto de aristas  
  
    var  $c$ : arista  
     $C := V - \{k\}$   
     $T := \{\}$   
    do  $n-1$  times  $\rightarrow$   
         $c :=$  arista  $\{i, j\}$  de costo mínimo tal que  $i \in C$  y  $j \notin C$   
         $C := C - \{i\}$   
         $T := T \cup \{c\}$   
    od  
end fun
```

donde  $n = |V|$ . La condición del ciclo podría reemplazarse por  $|T| < n - 1$  o  $C \neq \{\}$ , entre otras.

## Implementación del Algoritmo de Kruskal

```
fun Kruskal(G=(V,A) con costos en las aristas)
    ret T: conjunto de aristas
var i,j: vértice; u,v: componente conexa; c: arista
    C:= A
    T:= {}
do |T| < n - 1 → c:= arista {i,j} de C de costo mínimo
        C:= C-{c}
        u:= find(i)
        v:= find(j)
        if u ≠ v → T:= T ∪ {c}
            union(u,v)
        fi
od
end fun
```

## El problema union-find

Es el problema de cómo mantener un conjunto finito de elementos distribuidos en distintas componentes. Las operaciones que se quieren realizar son tres:

- init** inicializar diciendo que cada elemento está en una componente integrada exclusivamente por ese elemento,
- find** encontrar la componente en que se encuentra un elemento determinado,
- union** unir dos componentes para que pasen a formar una sola que tendrá la unión de los elementos que había en ambas componentes.



## El problema union-find

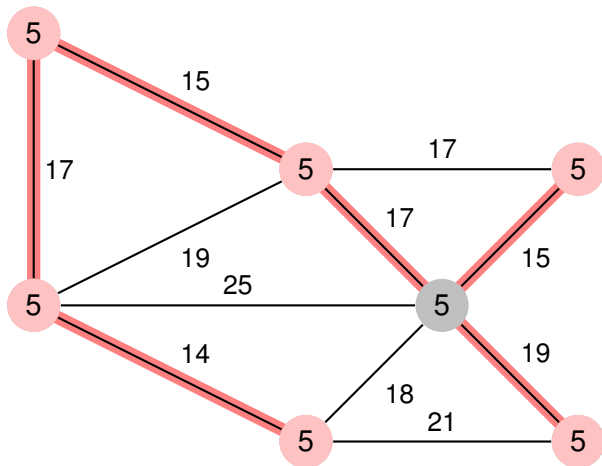
- De sólo manipularse por estas tres operaciones, las componentes serán siempre disjuntas
- y siempre tendremos que la unión de todas ellas dará el conjunto de todos los elementos.
- Una componente corresponde a una clase de equivalencia donde la relación de equivalencia sería “ $a \equiv b$  sii  $a$  y  $b$  pertenecen a la misma componente.”

## ¿cómo implementar una componente?

- Podemos pensar que una componente estará dada por un representante de esa componente.
- Esto permite implementarlas a través de una tabla que indica para cada elemento cuál es el representante de (la componente de) dicho elemento.
- Dado que asumimos una cantidad finita de elementos, los denotamos con números de 1 a  $n$ .
- La tabla que indica cuál es el representante de cada elemento será entonces un arreglo indexado por esos números:

```
type trep = array[1.. $n$ ] of nat
```

## Algoritmo de Kruskal, primer intento



## Primer intento

```
proc init(out rep: trep)  
    for i:= 1 to n do rep[i]:= i od  
end proc
```

```
fun find(rep: trep, i: nat) ret r: nat  
    r:= rep[i]  
end fun
```

```
{Pre:  $u \neq v \wedge u = \text{rep}[u] \wedge v = \text{rep}[v]$ }  
proc union(in/out rep: trep, in u,v: nat)  
    for i:= 1 to n do  
        if rep[i]=u  $\rightarrow$  rep[i]:= v fi  
    od  
end proc
```

# Primer intento

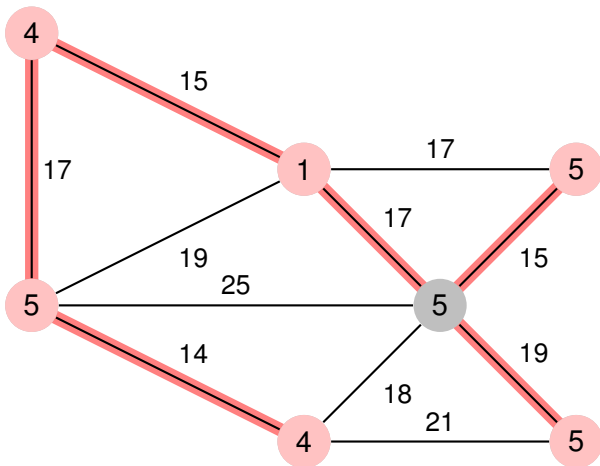
## Análisis

init es lineal

find es constante

union es lineal

## Algoritmo de Kruskal, segundo intento



## Segundo intento

```
fun is_rep(rep: trep, i: nat) ret b: bool  
    b:= (rep[i] = i)  
end fun  
  
{Pre:  $u \neq v \wedge \text{is\_rep}(\text{rep}, u) \wedge \text{is\_rep}(\text{rep}, v)$ }  
proc union(in/out rep: trep, in u,v: nat)  
    rep[u]:= v  
end proc  
  
fun find(rep: trep, i: nat) ret r: nat  
    var j: nat  
    j:= i;  
    do  $\neg \text{is\_rep}(\text{rep}, j) \rightarrow j:= \text{rep}[j]$  od  
    r:= j  
end fun
```

## Segundo intento

### Análisis

`init` es lineal

`find` es lineal en el peor caso

`union` es constante



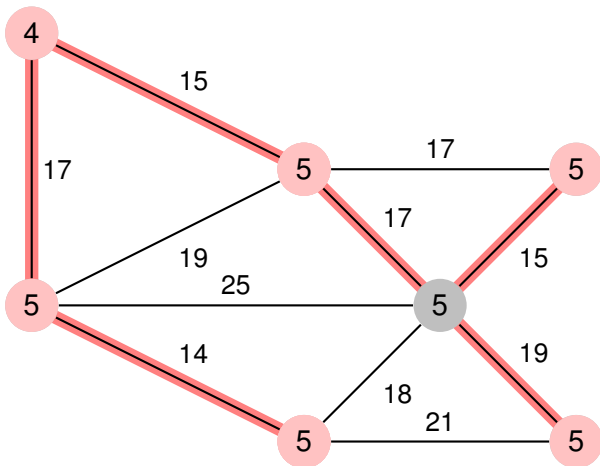
## Tercer intento

```
fun find(in/out rep: trep, i: nat) ret r: nat  
  var j,k: nat  
  j:= i  
  do  $\neg$  is_rep(rep,j)  $\rightarrow$  j:= rep[j] od  
  r:= j  
  j:= i  
  do  $\neg$  is_rep(rep,j)  $\rightarrow$  k:= rep[j]  
    rep[j]:= r  
    j:= k  
  
  od  
end fun
```

## Explicación

- Una vez calculado el representante  $r$ , la función `find` realiza una segunda recorrida desde  $i$  actualizando el arreglo `rep`.
- Tanto en la posición  $i$ , como en las posiciones de los  $j$  que fueron representantes de  $i$  se asigna directamente  $r$ .
- Esto vuelve constantes las futuras llamadas a `find(rep,i)` o `find(rep,j)`.
- Observar el uso **excepcional** de **in/out** asociado al parámetro `rep`.
- Esto se debe a que `find` no es estrictamente una función ya que modifica dicho parámetro.
- De todas formas se comporta como función ya que `find(rep,i) == find(rep,i)` siempre da verdadero.

## Algoritmo de Kruskal, tercer intento



# Tercer intento

## Análisis

- `init` es lineal
- `find` es lineal en el peor caso, pero se torna constante después de ejecutarse
- `union` es constante

## Último intento

```
proc init(out rep: trep)  
    for i:= 1 to n do rep[i]:= -1 od  
end proc  
  
fun is_rep(rep: trep, i: nat) ret b: bool  
    b:= (rep[i] < 0)  
end fun  
  
proc union(in/out rep: trep, in u,v: nat)  
    if rep[u]  $\geq$  rep[v]  $\rightarrow$  rep[v]:= rep[u]+rep[v]  
        rep[u]:= v  
    rep[u] < rep[v]  $\rightarrow$  rep[u]:= rep[u]+rep[v]  
        rep[v]:= u  
  
    fi  
end proc
```

## Último intento

Como vemos, ahora el arreglo debe permitir también números negativos:

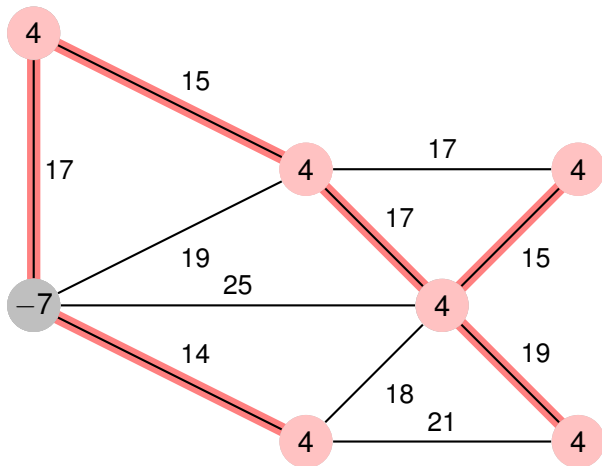
```
type trep = array[1..n] of int
```

## Último intento

Repetimos la definición de find para comprobar que sólo se efectúa  $j := \text{rep}[j]$  y  $k := \text{rep}[j]$  cuando  $\text{rep}[j]$  es un natural:

```
fun find(in/out rep: trep, i: nat) ret r: nat  
  var j,k: nat  
  j:= i  
  do  $\neg \text{is\_rep}(\text{rep},j) \rightarrow j := \text{rep}[j]$  od  
  r:= j  
  j:= i  
  do  $\neg \text{is\_rep}(\text{rep},j) \rightarrow$   
    k:= rep[j]  
    rep[j]:= r  
    j:= k  
  od  
end fun
```

## Algoritmo de Kruskal, último intento





# Último intento

## Análisis

`init` es lineal

`find` es en la práctica, es constante

`union` es constante