Algoritmos y Estructuras de Datos II

Algoritmos voraces

8 de mayo de 2019

Clase de hoy

Organización de la materia

- cómo vs. qué
- 3 partes
  1. análisis de algoritmos
  2. tipos de datos
  3. técnicas de resolución de problemas
     - divide y vencerás
     - algoritmos voraces
     - backtracking
     - programación dinámica
     - recorrida de grafos

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**, es decir, tenemos un problema que queremos resolver de manera **óptima**:
  - el camino más corto que une dos ciudades,
  - el valor máximo alcanzable entre ciertos objetos,
  - el costo mínimo para proveer un cierto servicio,
  - el menor número de billetes para pagar un cierto importe,
  - el menor tiempo necesario para realizar un trabajo, etc.
- 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 todos los problemas admiten solución voraz,
- pero varios problemas interesantes sí admiten solución voraz,
- y entonces, dichas soluciones resultan muy eficientes.

Problemas con solución voraz

- Problema de la moneda (casos especial).
- Problema de la mochila (caso especial).
- Problema del camino de costo mínimo en un grafo.
- Problema del árbol generador de costo mínimo en un grafo.
- Muchos otros problemas menos conocidos.

Ingredientes comunes de los algoritmos voraces

- se tiene un problema a resolver de manera **óptima**,
- un conjunto de **candidatos** a integrar la solución,
- los candidatos se van clasificando en 3: los aún no considerados, los **incorporados** a la solución parcial, y los **descartados**,
- existe una función que chequea si los candidatos incorporados ya forman una **solución** del problema (sin preocuparse por si la misma es o no óptima),
- una segunda función que comprueba si un conjunto de candidatos es **factible** de crecer hacia una solución (sin preocuparse por cuestiones de optimalidad),
- finalmente, una tercer función que **selecciona** de entre los candidatos aún no considerados, el más promisorio.

Receta general de los algoritmos voraces

- Inicialmente ningún candidato ha sido considerado, es decir, ni incorporado ni descartado.
- En cada paso se utiliza la función de **selección** para elegir cuál candidato considerar.
- Se utiliza la función **factible** para evaluar si el candidato considerado se incorpora a la solución o no.
- Se utiliza la función **solución** para comprobar si se ha llegado a una solución o si el proceso de construcción debe continuar.

Forma general

```
fun voraz(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.

Problema de la moneda

- Tenemos una cantidad infinita de monedas de cada una de las siguientes denominaciones:
  - 1 peso,
  - 50 centavos,
  - 25 centavos,
  - 10 centavos,
  - 5 centavos
  - y 1 centavo.
- Se desea pagar un cierto monto de manera exacta.
- Se debe determinar la manera de pagar dicho importe exacto con la menor cantidad de monedas posible.

Solución al problema de la moneda

- Seleccionar una moneda de la mayor denominación posible que no exceda el monto a pagar,
- utilizar exactamente el mismo algoritmo para el importe remanente.

Criterio de selección claramente establecido.

Algoritmo voraz

fun cambio(m: monto) **ret** S: conjunto de monedas
  var c, resto: monto
  C:= {100, 50, 25, 10, 5, 1}
  S:= {}
  resto:= m
  do resto $>$ 0 $\rightarrow$ c:= mayor elemento de C tal que c $\leq$ resto
    S:= S$\cup${una moneda de denominación c}
    resto:= resto - c
  od
end fun

Algoritmo voraz

Versión más parecida al esquema general de algoritmos voraces

```
fun cambio(m: monto) ret S: conjunto de monedas
    var c, resto: monto
    C:= {100, 50, 25, 10, 5, 1}
    S:= {}
    resto:= m
    do resto > 0 → c:= mayor elemento de C
                   C:= C - {c}
                   S:= S∪{resto div c  monedas de denominación c}
                   resto:= resto mod c
    od
end fun
```

Algoritmo voraz
Versión más detallada

```
fun cambio(m: monto) ret S: array[1..6] of nat
    var resto : monto
    resto := m
    S[1]:= resto div 100
    resto:= resto mod 100
    S[2]:= resto div 50
    resto:= resto mod 50
    S[3]:= resto div 25
    resto:= resto mod 25
    S[4]:= resto div 10
    resto:= resto mod 10
    S[5]:= resto div 5
    resto:= resto mod 5
    S[6]:= resto
end fun
```

Algoritmo voraz

Detallado pero genérico, asumiendo arreglo ordenado de denominaciones

{Pre: d[1] $\geq$ d[2] $\geq \ldots \geq$ d[n] }
fun cambio(d:**array**[1..n] **of nat**, m: monto) **ret** S: **array**[1..n] **of nat**
    var resto : monto
    resto := m
    for i:= 1 **to** n **do**
        S[i]:= resto div d[i]
        resto:= resto mod d[i]
    od
end fun

Sobre este algoritmo

- El orden del algoritmo es n, es decir, el número de denominaciones.
- Si el arreglo de denominaciones no está ordenado requiere $n \log n$ ordenarlo y luego n más el algoritmo, en total es $n \log n$.
- No siempre funciona, depende del conjunto de denominaciones.
- Para un conjunto razonable, funciona.

Conjunto de denominaciones para el que no funciona

- Sean 4, 3 y 1 las denominaciones y sea 6 el monto a pagar.
- El algoritmo voraz intenta pagar con una moneda de denominación 4, queda un saldo de 2 que solamente puede pagarse con 2 monedas de 1, en total, 3 monedas.
- Pero hay una solución mejor: dos monedas de 3.
- De todas formas, el algoritmo anda bien para todas las denominaciones de uso habitual.

Problema de la mochila

- Tenemos una mochila de capacidad W.
- Tenemos n objetos de valor $v_1, v_2, \ldots, v_n$ y peso $w_1, w_2, \ldots, w_n$.
- Se quiere encontrar la mejor selección de objetos para llevar en la mochila.
- Por mejor selección se entiende aquélla que totaliza el mayor valor posible sin que su peso exceda la capacidad W de la mochila.
- Para que el problema no sea trivial, asumimos que la suma de los pesos de los n objetos excede la capacidad de la mochila, obligándonos entonces a seleccionar cuáles cargar en ella.

Criterio de selección

¿Cómo conviene seleccionar un objeto para cargar en la mochila?

- El más valioso de todos.
- El menos pesado de todos.
- Una combinación de los dos.

Análisis del primer criterio de selección

El más valioso primero

- Razonabilidad: el objetivo es cargar la mochila con el mayor valor posible, escogemos los objetos más valiosos.
- Falla: puede que al elegir un objeto valioso dejemos de lado otro apenas menos valioso pero mucho más liviano.
- Ejemplo: Mochila de capacidad 10, objetos de valor 12, 11 y 9, y peso 7, 5 y 5.
- De elegir primero el de mayor valor (12) ocuparíamos 7 de los 10 kg de la mochila, no quedando lugar para otro objeto.
- En cambio, de elegir el de valor 11, ocuparíamos solamente 5 kg quedando 5 kg para el de valor 9, totalizando un valor de 20.

Análisis del segundo criterio de selección

El menos pesado primero

- Razonabilidad: hay que procurar aprovechar la capacidad de la mochila, escogemos los objetos más livianos.
- Falla: puede que al elegir un objeto liviano dejemos de lado otro apenas más pesado pero mucho más valioso.
- Ejemplo: Mochila de capacidad 13, objetos de valor 12, 11 y 7, y peso 6, 6 y 5.
- De elegir primero el de menor peso (5) obtendríamos su valor (7) más, en el mejor de los casos, 12, totalizando 12+7=19.
- En cambio, de elegir los dos de peso 6, no se excede la capacidad de la mochila y se totaliza un valor de 23.

Análisis del tercer criterio de selección

Combinando ambos criterios

- Debemos asegurarnos de que cada kg utilizado de la mochila sea aprovechado de la mejor manera posible: que cada kg colocado en la mochila valga lo más posible.
- Criterio: elegir el de mayor valor relativo (cociente entre el valor y el peso): dicho cociente expresa el valor promedio de cada kg de ese objeto.
- Falla: puede que al elegir un objeto dejemos de lado otro de peor cociente, pero que aprovecha mejor la capacidad.
- Ejemplo: Mochila de capacidad 10, objetos de valor 12, 11 y 8, y peso 6, 5 y 4.
- El critero elige al que pesa 5, ya que cada kg de ese objeto vale más de 2. Pero convenía elegir los otros dos.

Problema de la mochila

Versión simplificada

- El problema de la mochila no admite solución voraz.
- Se simplifica permitiendo **fraccionar** objetos.
- Ahora sí el tercer criterio funciona.
- (En el ejemplo anterior, elegimos primero el que vale 11 y luego 5/6 del que vale 12 obteniendo como valor total 11 + 10 = 21).

Algoritmo voraz

```
{Pre: ∑_{i=1}^{n} w[i] ≥ W}
fun mochila(v: array[1..n] of valor, w: array[1..n] of peso, W: peso)
                                          ret s: array[1..n] of real
    var resto: peso; c: nat
    for i:= 1 to n do s[i]:= 0 od
    resto:= W
    do resto > 0 → c:= tal que s[c] = 0 ∧ v[c]/w[c] máximo
                   if w[c] ≤ resto → s[c]:= 1
                                     resto:= resto - w[c]
                      w[c] > resto → s[c]:= resto/w[c]
                                     resto:= 0
                   fi
    od
```

Sobre este algoritmo

- Si los objetos ya están ordenados según su cociente valor/peso, el orden del algoritmo es n, es decir, el número de objetos.
- Si los objetos no están ordenado según su cociente valor/peso, requiere $n \log n$ ordenarlo y luego n más el algoritmo, en total es $n \log n$.
- Si los objetos en total exceden muy largamente la capacidad de la mochila, en vez de ordenar puede convenir utilizar una cola de prioridades, en cuyo caso el orden es n.
- funciona siempre que esté permitido fraccionar objetos.

Algoritmo voraz si los objetos están ordenados

{Pre: $\sum_{i=1}^{n} w[i] \geq W \wedge \forall i.v[i]/w[i] \geq v[i+1]/w[i+1]$}

```
fun mochila(v: array[1..n] of valor, w: array[1..n] of peso, W: peso)
                                          ret s: array[1..n] of real
    var resto: peso; c: nat
    for i:= 1 to n do s[i]:= 0 od
    resto:= W
    c:= 1
    do w[c] ≤ resto → s[c]:= 1
                      resto:= resto - w[c]
                      c:= c+1
    od
    s[c]:= resto/w[c]
end fun
```

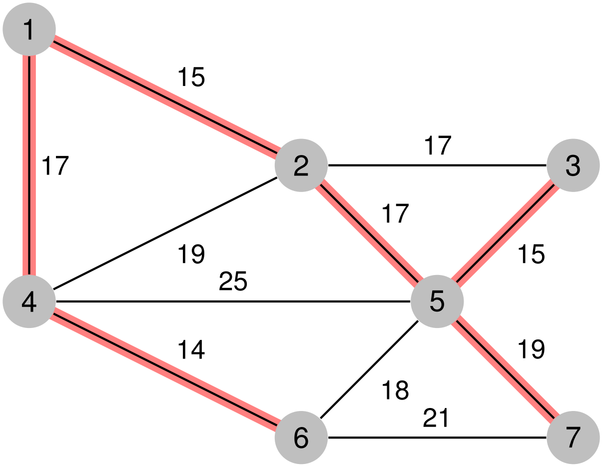
Árbol generador de costo mínimo

- 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.

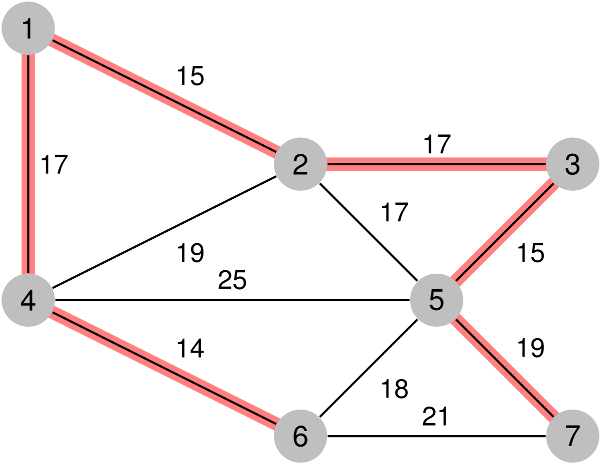
Árbol generador de costo mínimo

- 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

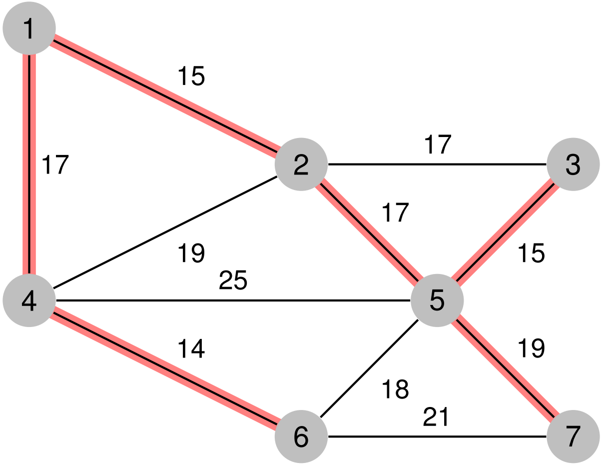


Dos estrategias

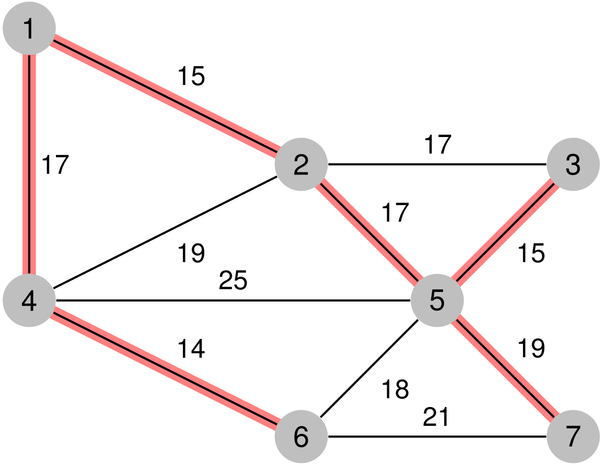
Hay dos grandes ideas de cómo resolverlo:

- 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
```

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

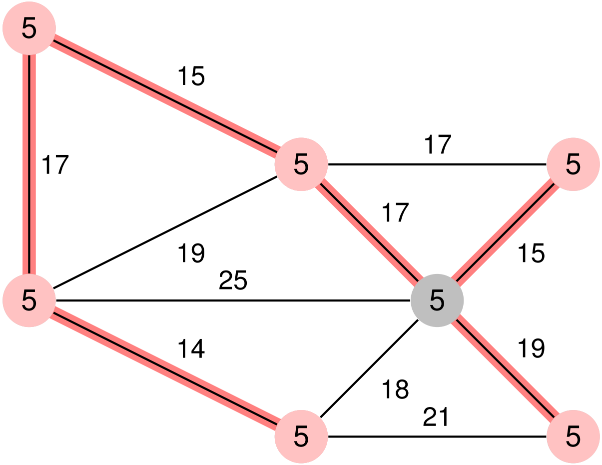
- De sólo manipularse por estas tres operaciones, las componentes serán siempre disjuntas
- y siempre tendremos que la unión de todas ellas dará el conjunto de todos los elementos.
- Una componente corresponde a una clase de equivalencia donde la relación de equivalencia sería "a $\equiv$ b sii a y b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

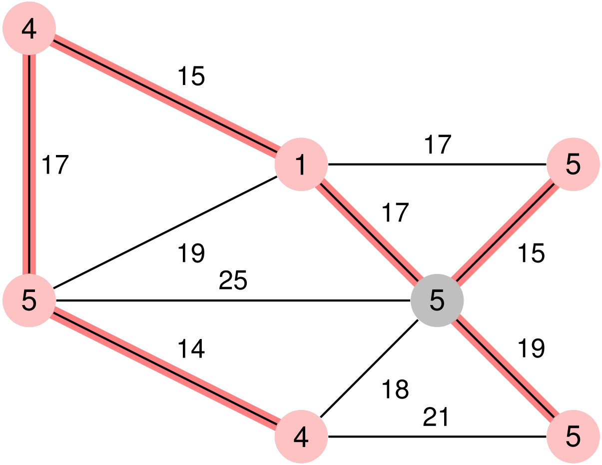
fun find(rep: trep, i: **nat**) **ret** r: **nat**
  r:= rep[i]
end fun

{Pre: $u \neq v \wedge u = rep[u] \wedge v = rep[v]$}
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: $u \neq v \wedge$ is_rep(rep,u) $\wedge$ is_rep(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 $\neg$ is_rep(rep,j) $\rightarrow$ j:= rep[j] **od**
    r:= j
end fun

Segundo intento

Análisis

init	es lineal
find	es lineal en el peor caso
union	es constante

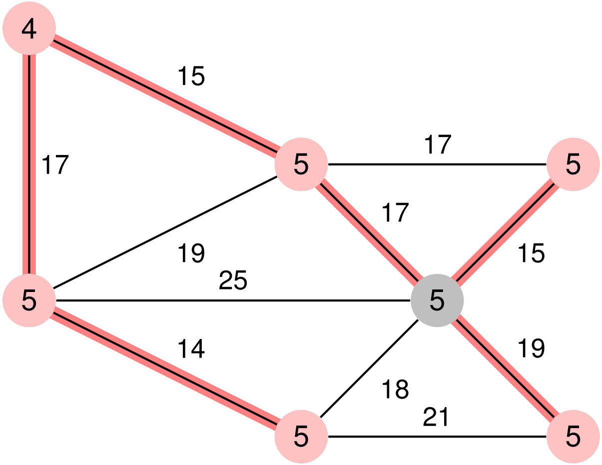
Tercer intento

```
fun find(in/out rep: trep, i: nat) ret r: nat
    var j,k: nat
    j:= i
    do ¬ is_rep(rep,j) → j:= rep[j] od
    r:= j
    j:= i
    do ¬ is_rep(rep,j) → 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] ≥ rep[v] → rep[v]:= rep[u]+rep[v]
                         rep[u]:= v
       rep[u] < rep[v] → rep[u]:= rep[u]+rep[v]
                         rep[v]:= u
    fi
```

Último intento

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

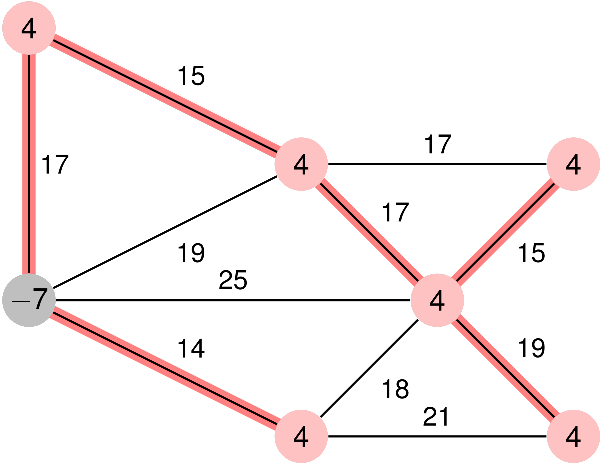
type trep = **array**[1..n] **of int**

Último intento

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

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

Algoritmo de Kruskal, último intento



Último intento

Análisis

init	es lineal
find	en la práctica, es constante
union	es constante

Camino de costo mínimo

- Sea $G = (V, A)$ un grafo dirigido con costos no negativos en sus aristas, y sea $v \in V$ uno de sus vértices.
- Se busca obtener los caminos de menor costo desde v hacia cada uno de los demás vértices.

Algoritmo de Dijkstra
Idea

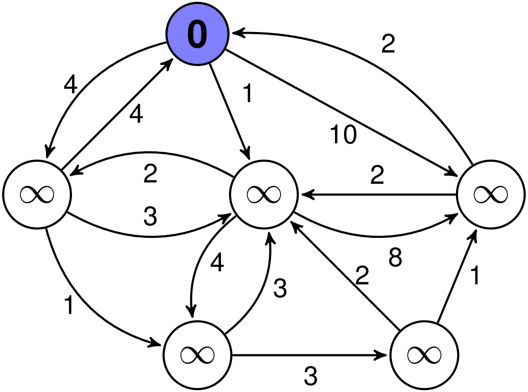
- El algoritmo de Dijkstra realiza una secuencia de n pasos, donde n es el número de vértices.
- En cada paso, “aprende” el camino de menor costo desde v a un nuevo vértice.
- A ese nuevo vértice lo pinta de azul.
- Tras esos n pasos, conoce los caminos de menor costo a cada uno de los vértices.

Algoritmo de Dijkstra
Ejemplo

- Tratemos de entenderlo a través de un ejemplo.
- En casa paso, en los vértices azules anotamos el costo del camino de menor costo de v a ese vértice.
- En casa paso, en los vértices blancos anotamos el costo del camino azul de menor costo de v a ese vértice.
- Un camino azul es uno que a lo sumo tiene al vértice destino blanco, sus otros vértices son azules.

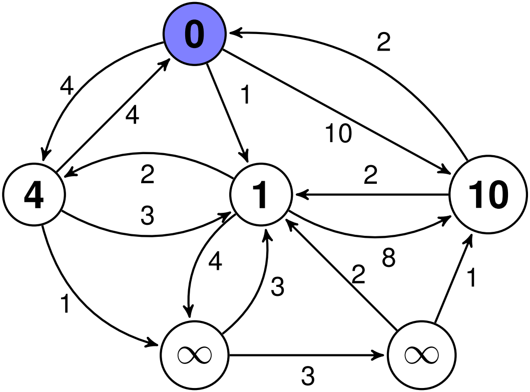
Algoritmo de Dijkstra

Paso 1 (a): sabemos lo que cuesta llegar de *v* a *v*



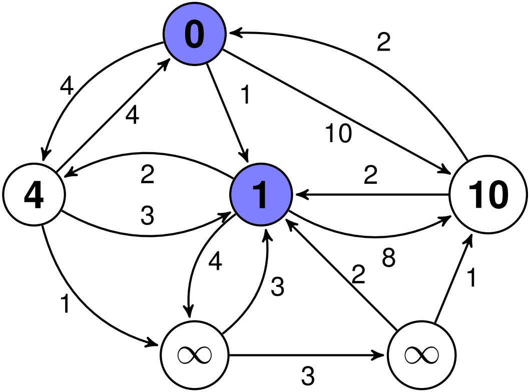
Algoritmo de Dijkstra

Paso 1 (b): Actualizamos los costos de los caminos azules óptimos



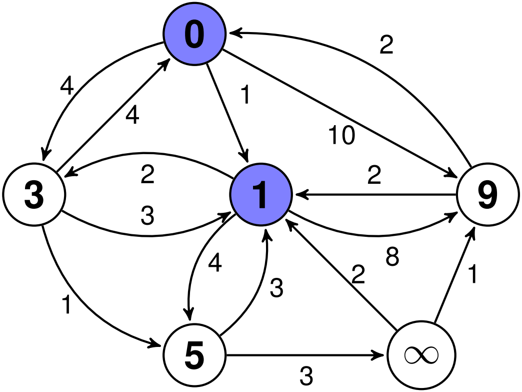
Algoritmo de Dijkstra

Paso 2 (a): sabemos lo que cuesta llegar de *v* a un nuevo vértice



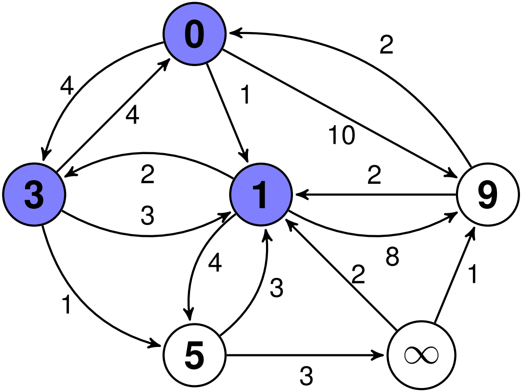
Algoritmo de Dijkstra

Paso 2 (b): Actualizamos los costos de los caminos azules óptimos



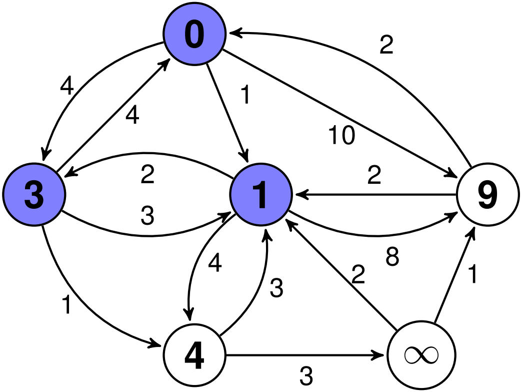
Algoritmo de Dijkstra

Paso 3 (a): sabemos lo que cuesta llegar de v a un nuevo vértice



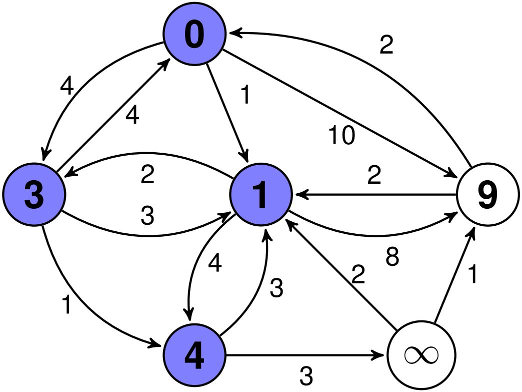
Algoritmo de Dijkstra

Paso 3 (b): Actualizamos los costos de los caminos azules óptimos



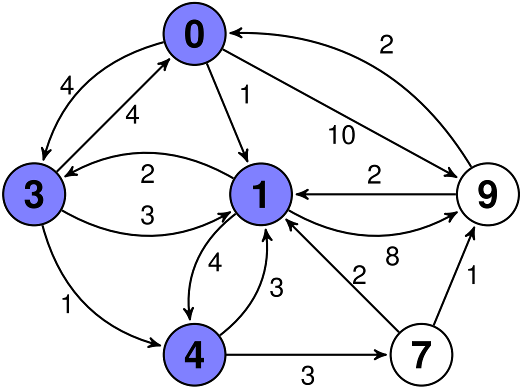
Algoritmo de Dijkstra

Paso 4 (a): sabemos lo que cuesta llegar de v a un nuevo vértice



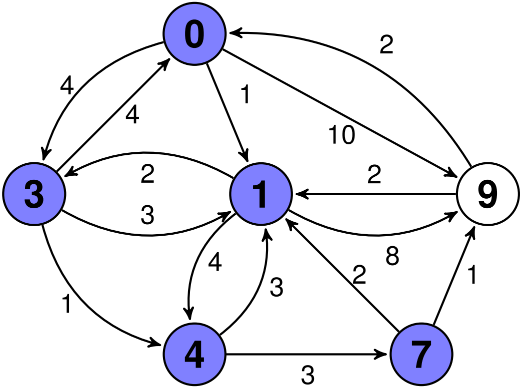
Algoritmo de Dijkstra

Paso 4 (b): Actualizamos los costos de los caminos azules óptimos



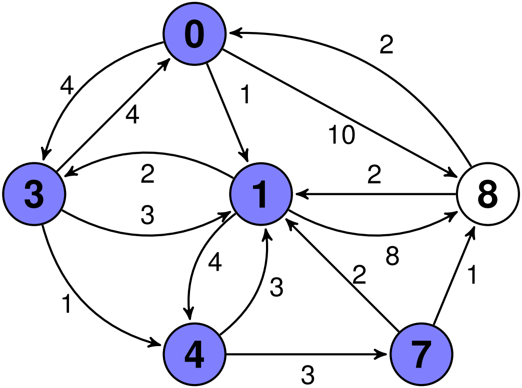
Algoritmo de Dijkstra

Paso 5 (a): sabemos lo que cuesta llegar de v a un nuevo vértice



Algoritmo de Dijkstra

Paso 5 (b): Actualizamos los costos de los caminos azules óptimos



El algoritmo

- Asumiremos que el grafo viene dado por el conjunto de vértices $V = \{1, 2, \dots, n\}$
- y los costos por una matriz L : **array**[1..n,1..n] **of** costo,
- que en $L[i, j]$ mantiene el costo de la arista que va de i a j.
- En caso de no haber ninguna arista de i a j, $L[i, j] = \infty$.
- Asumimos $L[j, j] = 0$.
- El algoritmo funciona también para grafos no dirigidos, simplemente se tiene $L[i, j] = L[j, i]$ para todo par de vértices i y j.

El algoritmo
Versión simplificada

- En vez de hallar el **camino de costo mínimo** desde v hasta cada uno de los demás, halla sólo el **costo** de dicho camino.
- Es decir, halla el **costo del camino de costo mínimo** desde v hasta cada uno de los demás.
- El resultado estará dado por un arreglo D: **array**[1..n] **of** costo,
- en $D[j]$ devolverá el costo del camino de costo mínimo que va de v a j.
- El conjunto C es el conjunto de los vértices hacia los que todavía desconocemos cuál es el camino de menor costo.

Algoritmo de Dijkstra

```
fun Dijkstra(L: array[1..n,1..n] of costo, v: nat)
                                        ret D: array[1..n] of costo
    var c: nat
    C:= {1,2,...,n}-{v}
    for j:= 1 to n do D[j]:= L[v,j] od
    do n-2 times → c:= elemento de C que minimice D[c]
                   C:= C-{c}
                   for j in C do D[j]:= min(D[j],D[c]+L[c,j]) od
    od
end fun
```

Vértices azules

- Llamamos **vértices azules** a los que no pertenecen a C.
- O sea, a los pintados de azul en nuestra animación anterior.
- Inicialmente el único vértice azul es v.
- Un **camino azul** es un camino cuyos vértices son azules salvo quizá el último.
- Inicialmente, los caminos azules son el camino vacío (que va de v a v y tiene costo $L[v, v] = 0$)
- y las aristas que van de v a j que tienen costo $L[v, j]$.

Idea del algoritmo

- En todo momento, D mantiene en cada posición j, el costo del camino **azul** de costo mínimo que va de v a j.
- Inicialmente, por lo dicho en el párrafo anterior, $D[j]$ debe ser $L[v, j]$.
- Eso explica la inicialización de D que se realiza en el primer **for**.

Vértice azul y camino mínimo

- Cuando un vértice c es azul, ya se conoce el costo del camino de costo mínimo que va de v a c,
- y es el que está dado en ese momento por $D[c]$.
- En efecto, esto se cumple inicialmente: el vértice v es el único azul y el valor inicial de $D[v]$, es decir, 0, es el costo del camino de costo mínimo para ir desde v a v.

Invariante

Lo dicho puede expresarse en el siguiente invariante:

$\forall j \notin C.D[j] =$ costo del camino de costo mínimo de v a j
$\forall j \in C.D[j] =$ costo del camino **azul** de costo mínimo de v a j

- Para entender el algoritmo es importante prestar atención a la palabra **azul**.
- Cuando conocemos el costo del camino **azul** de costo mínimo no necesariamente hemos obtenido lo que buscamos,
- buscamos el costo del camino de costo mínimo, el mínimo de todos, azul o no.

Un nuevo vértice azul

- El algoritmo de Dijkstra elimina en cada ciclo un vértice c de C.
- Para que se mantenga el invariante es imprescindible saber que para ese c
- (que pertenecía a C y por lo tanto por el invariante $D[c]$ era el costo del camino **azul** de costo mínimo de v a c),
- $D[c]$ es en realidad el costo del camino (no necesariamente azul) de costo mínimo de v a c.

¿Cómo podemos asegurarnos de eso?

- El algoritmo elige $c \in C$ de modo de que $D[c]$ sea el mínimo.
- Es decir, elige un vértice c que aún **no es azul** y tal que $D[c]$ es mínimo.
- Sabemos, por el invariante, que $D[c]$ es el costo del camino **azul** de costo mínimo de v a c.
- ¿Puede haber un camino **no azul** de v a c que cueste menos?

¿Puede haber un camino **no azul** de v a c que cueste menos?

- Si lo hubiera, dicho camino necesariamente debería tener, por ser **no azul**, algún vértice intermedio **no azul**.
- Sea w el primer vértice **no azul** que ocurre en ese camino comenzando desde v.
- El camino **no azul** consta de una primera parte que llega a w.
- Esa primera parte es un camino **azul** de v a w, por lo que su costo, dice el invariante, debe ser $D[w]$.
- El costo del camino completo **no azul** de v a c que pasa por w costará al menos $D[w]$ ya que ése es apenas el costo de una parte del mismo.

¿Puede haber un camino **no azul** de v a c cueste menos?

- Dijimos que ese camino pasaría por w como primer vértice **no azul** y por ello costaría al menos $D[w]$.
- Sin embargo, como c fue elegido como el que minimiza (entre los vértices **no azules**) $D[c]$, necesariamente debe cumplirse $D[c] \leq D[w]$.
- Esto demuestra que no puede haber un camino **no azul** de v a c que cueste menos que $D[c]$.
- Por ello, c puede sin peligro ser considerado un vértice **azul** ya que $D[c]$ contiene el costo del camino (azul o no) de costo mínimo de v a c.

Nuevos caminos azules

- Inmediatamente después de agregar c entre los vértices **azules**, es decir, inmediatamente después de eliminarlo de C,
- surgen nuevos caminos **azules** ya que ahora se permite que los mismos pasen también por el nuevo vértice **azul** c.
- Eso obliga a actualizar $D[j]$ para los j **no azules** de modo de que siga satisfaciendo el invariante.
- Ahora un camino **azul** a j puede pasar por c.
- Sólo hace falta considerar caminos **azules** de v a j cuyo último vértice **azul** es c.

¿Por qué?

- Dijimos que sólo hace falta considerar caminos **azules** de v a j cuyo último vértice **azul** es c.
- ¿Por qué?
- Los caminos **azules** de v a j que pasan por c y cuyo último vértice **azul** es k no ganan nada por pasar por c
- ya que c está antes de k en esos caminos y entonces el costo del tramo hasta k, siendo k **azul**, sigue siendo como mínimo $D[k]$,
- es decir, en el mejor de los casos lo mismo que se tenía sin pasar por c.

Recalculando D

- Consideremos entonces solamente los caminos **azules** a j que tienen a c como último vértice **azul**.
- El costo de un tal camino de costo mínimo está dado por $D[c] + L[c, j]$,
- la suma entre el costo del camino de costo mínimo para llegar hasta c ($D[c]$) más el costo de la arista que va de c a j ($L[c, j]$).
- Este costo debe compararse con el que ya se tenía, el que sólo contemplaba los caminos **azules** antes de que c fuera **azul**.
- Ese valor es $D[j]$.
- El mínimo de los dos es el nuevo valor para $D[j]$.
- Eso explica el segundo **for**.

Últimas consideraciones

- Por último, puede observarse que en cada ejecución del ciclo un nuevo vértice se vuelve **azul**.
- Inicialmente v lo es.
- Por ello, al cabo de n-2 iteraciones, tenemos solamente 1 vértice **no azul**.
- Sea k ese vértice.

Postcondición

El invariante resulta

$$\forall j \neq k. D[j] = \text{costo del camino de costo mínimo de } v \text{ a } j$$
$$D[k] = \text{costo del camino } \textbf{azul} \text{ de costo mínimo de } v \text{ a } k$$

pero siendo k el único vértice **no azul** todos los caminos de v a k (que no tengan ciclos en los que k esté involucrado) son **azules.** Por ello, se tiene

$$D[k] = \text{costo del camino de costo mínimo de } v \text{ a } k$$

y por consiguiente

$$\forall j. D[j] = \text{costo del camino de costo mínimo de } v \text{ a } j$$

Algoritmo de Dijkstra

```
fun Dijkstra(L: array[1..n,1..n] of costo, v: nat)
    ret D: array[1..n] of costo                ret E: array[1..n] of nat
    var c: nat
    C:= {1,2,...,n}-{v}
    for j:= 1 to n do D[j]:= L[v,j] od
    for j:= 1 to n do E[j]:= v od
    do n-2 times → c:= elemento de C que minimice D[c]
                   C:= C-{c}
                   for j in C do
                       if D[c]+L[c,j] < D[j] then D[j]:= D[c]+L[c,j]
                                                  E[j]:= c
                       fi
                   od
    od
end fun
```

Algoritmo de Dijkstra

¿Cuál es el orden de este algoritmo?

```
fun Dijkstra(L: array[1..n,1..n] of costo, v: nat)
                                        ret D: array[1..n] of costo
    var c: nat
    C:= {1,2,. . . ,n}-{v}
    for j:= 1 to n do D[j]:= L[v,j] od
    do n-2 times → c:= elemento de C que minimice D[c]
                   C:= C-{c}
                   for j in C do D[j]:= min(D[j],D[c]+L[c,j]) od
    od
end fun
```

Respuesta: n^2. La versión que devuelve además el camino, también.

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.

Algoritmo de Prim en detalle ($L[x, y] = L[y, x]$)

fun Prim(L: **array**[1..n,1..n] **of** costo, v: **nat**) **ret** T: conjunto de aristas
    var D: **array**[1..n] **of** costo   **var** E: **array**[1..n] **of nat**
    var c: **nat**
    C:= {1,2,. . . ,n}-{v}   T:= {}
    for j:= 1 **to** n **do** D[j]:= L[v,j] **od**
    for j:= 1 **to** n **do** E[j]:= v **od**
    do n-1 **times** $\rightarrow$ c:= elemento de C que minimice D[c]
                        C:= C - {c}   T:= T $\cup$ {(E[c],c)}
                        for j **in** C **do**
                            if L[c,j] < D[j] **then** D[j]:= L[c,j]
                                                E[j]:= c
                            fi
                        od
    od
end fun

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 $\rightarrow$ c:= arista $\{i, j\}$ de C de costo mínimo
        C:= C-{c}
        u:= find(i)
        v:= find(j)
        if u $\neq$ v $\rightarrow$ T:= T $\cup$ {c}
            union(u,v)
        fi
    od
end fun

Conclusión

Sea n el número de vértices de un grafo.

- El algoritmo de Dijkstra es del orden de n^2.
- El algoritmo de Dijkstra que devuelve también el camino, es del orden de n^2.
- El algoritmo de Prim es del orden de n^2.
- El algoritmo de Kruskal es del orden de $n^2 * \log n$.
- En principio son buenos órdenes, un grafo puede tener del orden de n^2 aristas.
- Cuando el grafo tiene mucho menos de n^2 aristas, en general todos estos algoritmos pueden reescribirse de modo de que su orden mejore.