Algoritmos voraces

Yolanda Ortega Mallén

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Sumario

- Un ejemplo: el problema de la mochila.
- Esquema general.
- Aplicaciones.

Problema de la mochila

Maximizar el valor total de los objetos metidos en la mochila.

Problema de la mochila: planteamiento

Datos: n objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$.
Los objetos son fraccionables.
La mochila soporta un peso total máximo $M > 0$.

Objetivo: Maximizar $\sum_{i=1}^{n} x_i v_i$ con la restricción $\sum_{i=1}^{n} x_i p_i \leq M$,
x_i es la fracción del objeto i metida en la mochila, $0 \leq x_i \leq 1$.

Suponemos $\sum_{i=1}^{n} p_i > M$.

La solución óptima deberá llenar la mochila por completo: $\sum_{i=1}^{n} x_i p_i = M$.

Problema de la mochila: estrategia

Si un objeto es bueno, cuanto más cantidad metamos en la mochila mejor.

fun `mochila-v1`$(P[1..n], V[1..n]$ **de** $real^+, M : real^+)$ **dev** $sol[1..n]$ **de** $real$
    $sol[1..n] := [0]$
    $peso := 0$
    mientras $peso < M$ **hacer**
        $i :=$ el mejor objeto restante
        si $peso + P[i] \leq M$ **entonces**
            $sol[i] := 1$
            $peso := peso + P[i]$
        si no
            $sol[i] := (M - peso)/P[i]$
            $peso := M$
        fsi
    fmientras
ffun

Problema de la mochila: selección de objetos

¿Cuál es el mejor objeto restante?

1. Objeto más valioso: incrementar el valor total lo más rápido posible.
2. Objeto más ligero: agotar el peso lentamente para que quepan más objetos.

$M = 100$, $n = 5$

	1	2	3	4	5
p_i	10	20	30	40	50
v_i	20	30	66	40	60
$\frac{v_i}{p_i}$	2	1,5	2,2	1	1,2

Seleccionar	x_i					valor
máx v_i	0	0	1	0,5	1	146
mín p_i	1	1	1	1	0	156
máx $\frac{v_i}{p_i}$	1	1	1	0	0,8	164

1. Objetos muy valiosos pero muy pesados.
2. Objetos muy ligeros pero poco valiosos.

Solución: maximizar la relación valor por unidad de peso.

Problema de la mochila: Demostración de optimalidad

Objetos ordenados en forma decreciente respecto a su valor por unidad de peso:

$$\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \ldots \geq \frac{v_n}{p_n},$$

y sea $X = (x_1, x_2, \ldots, x_n)$ la solución construida por el algoritmo voraz.

Si todos los x_i son 1, la solución es óptima.

En caso contrario, sea j el menor índice tal que $x_j < 1$.
El algoritmo asegura que $x_i = 1$ si $i < j$ y $x_i = 0$ si $i > j$.

Sea $V(X) = \sum_{i=1}^{n} x_i v_i$ el valor total de la solución X.

Comparamos la solución X con una solución Y cualquiera:

$$V(X) - V(Y) = \sum_{i=1}^{n} (x_i - y_i) v_i = \sum_{i=1}^{n} (x_i - y_i) p_i \frac{v_i}{p_i}.$$

$i < j$ se tiene $x_i = 1$ y, por tanto, $x_i - y_i \geq 0$. Además $\frac{v_i}{p_i} \geq \frac{v_j}{p_j}$.
$i > j$ se tiene $x_i = 0$ y, por tanto, $x_i - y_i \leq 0$. Además $\frac{v_i}{p_i} \leq \frac{v_j}{p_j}$.
$i = j$ es obvio que $\frac{v_i}{p_i} = \frac{v_j}{p_j}$.

$\forall i : 1 \leq i \leq n$, se tiene que

$$(x_i - y_i)\frac{v_i}{p_i} \geq (x_i - y_i)\frac{v_j}{p_j}$$

y por tanto,

$$V(X) - V(Y) \geq \sum_{i=1}^{n} (x_i - y_i)p_i\frac{v_j}{p_j} = \frac{v_j}{p_j}\sum_{i=1}^{n}(x_i - y_i)p_i.$$

Ahora bien, $\frac{v_j}{p_j} > 0$ (todos los valores y pesos son positivos).

Por otra parte,

$$\sum_{i=1}^{n} x_i p_i = M \ \wedge \ \sum_{i=1}^{n} y_i p_i \leq M$$

Por lo que finalmente

$$V(X) - V(Y) \geq \frac{v_j}{p_j}\sum_{i=1}^{n}(x_i - y_i)p_i = \frac{v_j}{p_j}(\sum_{i=1}^{n} x_i p_i - \sum_{i=1}^{n} y_i p_i) \geq 0$$

Problema de la mochila: Algoritmo voraz

fun `mochila-v2`$(P[1..n], V[1..n]$ **de** $real^+, M : real^+)$ **dev** $\langle sol[1..n]$ **de** $real, valor : real\rangle$
var $D[1..n]$ **de** $real^+, I[1..n]$ **de** $1..n$ { para ordenar los índices }
  para $i = 1$ **hasta** n **hacer**
    $D[i] := P[i]/V[i]$
  fpara
  $I :=$ `ordenar-índices`(D)
  $peso := 0$; $i := 1$; $valor := 0$
  mientras $i \leq n \ \wedge_c \ peso + P[I[i]] \leq M$ **hacer**
    { podemos coger el objeto $I[i]$ entero }
    $sol[I[i]] := 1$; $peso := peso + P[I[i]]$
    $valor := valor + V[I[i]]$
    $i := i + 1$
  fmientras
  si $i \leq n$ **entonces** { partir el objeto $I[i]$ }
    $sol[I[i]] := (M - peso)/P[I[i]]$
    $valor := valor + V[I[i]] * sol[I[i]]$
    para $j = i + 1$ **hasta** n **hacer**
      $sol[I[j]] := 0$
    fpara
  fsi
ffun

Coste: ordenación de índices: $\Theta(n \log n)$; inicialización + bucle voraz: $\Theta(n)$.
Total: $\Theta(n \log n)$.

Características generales

Conjunto de candidatos para construir la solución.

A medida que avanza el algoritmo se van formando dos conjuntos:

- candidatos seleccionados: formarán parte de la solución,
- candidatos rechazados definitivamente.

Función de selección indica cuál es el candidato más prometedor de entre los aún no considerados.

Test de factibilidad comprueba si un candidato es compatible con la solución parcial construida hasta el momento.

Test de solución determina si una solución parcial forma una solución *completa*.

Función objetivo asocia un valor a cada solución (problemas de optimización).

Funcionamiento general

- Optimización local. Seleccionar en cada etapa el *mejor* según cierto criterio, sin tener en cuenta elecciones previas/futuras.
- Añadir a los seleccionados el candidato devuelto por la función de selección si se sigue cumpliendo el test de factibilidad. Si no, rechazarlo.
- Cada candidato se trata una sola vez. Las decisiones tomadas nunca se reconsideran.
- Los algoritmos son sencillos y eficientes.
- Importante: demostración de corrección.
- Método de reducción de diferencias: comparar una solución óptima con la solución obtenida por el algoritmo voraz. Si no son iguales, transformar la solución óptima de forma que continúe siendo óptima, pero más parecida a la del algoritmo voraz.

Esquema de algoritmo voraz

fun `voraz`$(datos : conjunto)$ **dev** $S : conjunto$
var $candidatos : conjunto$
    $S := \emptyset$ { en S se va construyendo la solución }
    $candidatos := datos$
    mientras $candidatos \neq \emptyset \wedge \neg$`es-solución?`$(S)$ **hacer**
        $x :=$ `seleccionar`$(candidatos)$
        $candidatos := candidatos - \{x\}$
        si `es-factible?`$(S \cup \{x\})$ **entonces** $S := S \cup \{x\}$ **fsi**
    fmientras
ffun

Aplicaciones en móviles

Tenemos n aplicaciones $A_1, A_2, \ldots, A_n$ que queremos instalar en nuestro móvil. La aplicación A_i ocupa $m_i\,MB$ y nuestro móvil tiene $M\,MB$ disponibles. Se quiere maximizar el número de aplicaciones almacenadas en el móvil.

Supondremos que $M < \sum_{i=1}^{n} m_i$.

Considerar las aplicaciones ordenadas de menor a mayor tamaño:

- si la aplicación cabe todavía en la memoria, instalarlo y pasar a la siguiente;
- si la aplicación no cabe, descartarla y terminar.

$\{\ m[1] \leq m[2] \leq \ldots \leq m[n]\ \wedge\ \sum_{i=1}^{n} m[i] > M\ \}$
fun `aplicaciones-en-móvil`$(m[1..n]$ **de** *real*$^+, M$: *real*$)$ **dev** $X[1..n]$ **de** $\{0,1\}$
    $X := [0]$; *acumula* $:= 0$; $i := 1$
    mientras *acumula* $+\, m[i] \leq M$ **hacer**
        $X[i] := 1$
        acumula $:=$ *acumula* $+\, m[i]$
        $i := i+1$
    fmientras
ffun

Aplicaciones en móviles: corrección

La estrategia conduce siempre a una solución óptima.

Solución cualquiera $Z = (z_1, z_2, \ldots, z_n)$:

$z_i = 0$ la aplicación A_i no se instala,
$z_i = 1$ A_i forma parte de la solución.

Número de aplicaciones instaladas: $\sum_{i=1}^{n} z_i$.

Comparamos la solución de la estrategia voraz, X, con una solución óptima Y.

La estrategia voraz escoge solo las k primeras aplicaciones:
$\forall i : 1 \leq i \leq k : x_i = 1$ y $\forall i : k < i \leq n : x_i = 0$.

Empezando a comparar (de izquierda a derecha) con Y, sea $j \geq 1$ la primera posición donde $x_j \neq y_j$:

$$\begin{array}{cccccccccc} 1 & 1 & \ldots & 1 & \ldots & 1 & 0 & \ldots & 0 \\ x_1 & x_2 & \ldots & x_j & \ldots & x_k & x_{k+1} & \ldots & x_n \\ = & = & \ldots & \neq & & & & & \\ y_1 & y_2 & \ldots & y_j & \ldots & y_k & y_{k+1} & \ldots & y_n \end{array}$$

Si $y_j \neq x_j = 1$, tenemos $y_j = 0$, y $\sum_{i=1}^{j} y_i = j - 1 < j = \sum_{i=1}^{j} x_i$.

Como Y es óptima, $\sum_{i=1}^{n} y_i \geq \sum_{i=1}^{n} x_i = k$. Existe $l > k \geq j$ tal que $y_l = 1$.

$$\begin{array}{ccccccccccc}
1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
= & = & \dots & \neq & & & & & \neq & & \\
y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
\end{array}$$

Como las aplicaciones están ordenadas por tamaño, $m_j \leq m_l$; y si A_l cabe en el móvil, podemos poner en su lugar A_j sin sobrepasar la capacidad total.

Con el cambio obtenemos Y' con $y'_j = 1 = x_j$, $y'_l = 0$ y para el resto $y'_i = y_i$. Esta nueva solución es más parecida a X, y tiene el mismo número de aplicaciones que Y (sigue siendo óptima).

Repitiendo el proceso, vamos igualando las aplicaciones en la solución óptima a las de la solución voraz X, hasta alcanzar la posición k.

Atención a pacientes: tiempo medio mínimo

En un centro de salud n pacientes llegan simultáneamente a la consulta de un médico; cada paciente va a necesitar un tiempo de atención t_i.
Se desea minimizar el tiempo medio de estancia de cada paciente en el centro de salud, esto es, el tiempo transcurrido desde que ha llegado a la consulta y termina de ser atendido.

El problema consiste en minimizar

$$TM = \frac{1}{n} \sum_{i=1}^{n} T_i$$

donde T_i es el tiempo en el centro de salud del paciente i.

Como n está fijo, esto es equivalente a minimizar el tiempo total

$$T = \sum_{i=1}^{n} T_i.$$

T_i = tiempo de atención (t_i) + tiempos de los pacientes atendidos antes

Ejemplo: $n = 3$, $t_1 = 5$, $t_2 = 10$, $t_3 = 3$

Orden de atención: 1, 2, 3

$$T = 5 + (5 + 10) + (5 + 10 + 3) = 38$$

Orden de atención: 1, 3, 2

$$T = 5 + (5 + 3) + (5 + 3 + 10) = 31$$

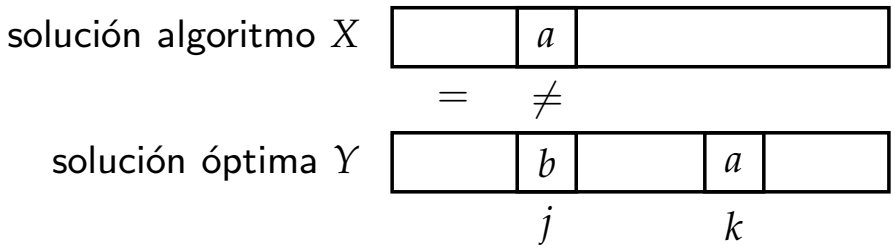
Orden de atención: 3, 1, 2

$$T = 3 + (3 + 5) + (3 + 5 + 10) = 29$$

Planificación óptima: atender a los pacientes por orden creciente de tiempo de atención.

Sea $I = i_1, i_2, \ldots, i_n$ una permutación cualquiera de los enteros de 1 a n.

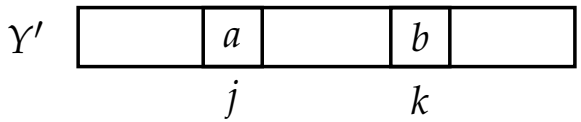
$$\begin{aligned} T(I) &= t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \cdots + (t_{i_1} + t_{i_2} + \cdots + t_{i_n}) \\ &= nt_{i_1} + (n-1)t_{i_2} + \cdots + 2t_{i_{n-1}} + t_{i_n} \\ &= \sum_{k=1}^{n} (n-k+1)t_{i_k}. \end{aligned}$$



Las soluciones son iguales hasta la posición j. Como el algoritmo atiende a los pacientes en orden creciente de tiempo de atención, $t_a \leq t_b$.

Existe una posición $k > j$ donde la solución óptima Y atiende al paciente a.

Modificamos la solución Y intercambiando los pacientes de las posiciones j y k:



Y' sigue siendo solución (permutación de los pacientes), ¿sigue siendo óptima?

$$T(Y) = \left(\sum_{\substack{l=1 \\ l \neq j,k}}^{n} (n-l+1)t_{y_l} \right) + (n-j+1)t_b + (n-k+1)t_a$$

$$T(Y') = \left(\sum_{\substack{l=1 \\ l \neq j,k}}^{n} (n-l+1)t_{y_l} \right) + (n-j+1)t_a + (n-k+1)t_b$$

$$\begin{aligned} T(Y) - T(Y') &= (n-j+1)t_b + (n-k+1)t_a - (n-j+1)t_a - (n-k+1)t_b \\ &= -jt_b + kt_b - kt_a + jt_a \\ &= t_b(k-j) - t_a(k-j) \\ &= (k-j)(t_b - t_a) \\ &\geq 0 \end{aligned}$$

Y' es también óptima.

Pokémon Go

¡Zeraora, Charmander, Bulbasaur, Pikachu y otros Pokémon se han escondido por nuestra Facultad! Ahora es tu oportunidad de descubrirlos y capturarlos. Cada Pokémon tiene un valor (positivo) asociado y estará disponible para ser capturado solo durante unos cuantos días, pasado este periodo, el Pokémon desaparecerá para siempre.
Teniendo en cuenta que capturar un Pokémon exige un día completo de búsqueda, tienes que decidir cuáles Pokémon capturar y cuándo hacerlo para maximizar el valor total.

Pokémon Go: Estrategia voraz

Cada Pokémon lleva asociado un valor $v_i > 0$ que solo podrá contabilizarse si el Pokémon es capturado sin superar el plazo correspondiente p_i.

Una secuencia de captura $(i_1, i_2, \ldots, i_k)$ de un conjunto de Pokémon es admisible si el plazo correspondiente a cada Pokémon es posterior al momento de su captura:

$$\forall j : 1 \leq j \leq k : p_{i_j} \geq j.$$

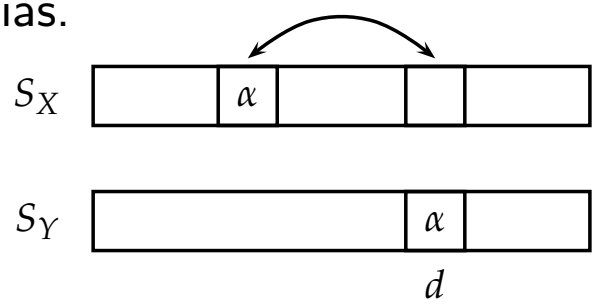
Un conjunto de Pokémon es factible si existe alguna secuencia de captura admisible.

Estrategia voraz: cada día se escoge el Pokémon con mayor valor de forma que el subconjunto formado siga siendo factible.

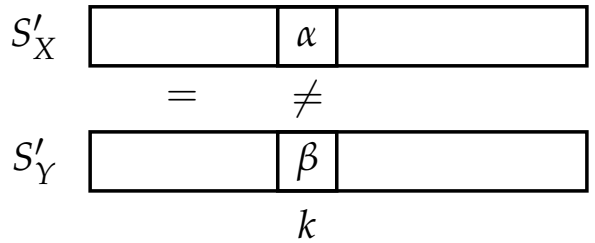
Pokémon Go: corrección

Sean X la solución del algoritmo voraz e Y una solución óptima.
Sean S_X y S_Y secuencias admisibles de los Pokémon de los subconjuntos X e Y.

Primero transformamos S_X e S_Y de tal forma que los Pokémon comunes se capturen en los mismos días.



Comparamos ahora las secuencias S'_X y S'_Y



Si no se captura ningún pokémon el día k en S'_X (S'_Y) se considera $v_\alpha = 0$ ($v_\beta = 0$).

$v_\alpha > v_\beta$ imposible, mejoraríamos Y.
$v_\alpha < v_\beta$ imposible, el algoritmo habría considerado β antes que α.
$v_\alpha = v_\beta$ se puede cambiar β por α en Y.

Pokémon Go: comprobación de la factibilidad

Para determinar si un conjunto de Pokémon es factible, basta con encontrar una secuencia de captura admisible.

Lema 1

Un conjunto P de Pokémon es factible si y solo si la secuencia de los Pokémon de P ordenados de forma creciente según plazo es admisible.

Demostración:

$(\Leftarrow)$ Trivial.
$(\Rightarrow)$ Por contrarrecíproco.
Sea $P = \{P_1, \ldots, P_k\}$ con $p_1 \leq p_2 \leq \ldots \leq p_k$,
tal que la secuencia $P_1, P_2, \ldots, P_k$ no es admisible,
es decir, existe algún Pokémon $P_r, r \in \{1, \ldots, k\}$, tal que $p_r < r$.

Pero entonces se cumple que $p_1 \leq p_2 \leq \ldots \leq p_{r-1} \leq p_r \leq r - 1$,
es decir, hay r Pokémon cuyos plazos son menores o iguales que $r - 1$, y por tanto es imposible capturar todos los Pokémon dentro de su plazo.

El conjunto P no es factible.

$\{\ V[1] \geq V[2] \geq \ldots \geq V[n]\ \}$
fun `Pokémon-v1`$(P[1..n], V[1..n]$ **de** $nat^+)$ **dev** $\langle\ Pokémon[1..n]$ **de** $1..n, k : 1..n\ \rangle$
$\{\ Pokémon[i]$ es el Pokémon capturado en el día i, para i entre 1 y $k\ \}$
$\{\ k$ es el número total de Pokémon capturados $\}$
var $aux[1..n]$ **de** $1..n$ $\{$ secuencia auxiliar, ordenada por plazos $\}$
$k := 1\ ;\ aux[1] := 1$ $\{$ el primer Pokémon siempre se captura $\}$
para $i = 2$ **hasta** n **hacer**
$\{$ buscar hueco $\}$
$d := k$
mientras $d > 0\ \wedge_c\ (P[aux[d]] > P[i]\ \wedge\ P[aux[d]] > d)$ **hacer**
$d := d - 1$
fmientras
$\{\ d = 0\ \vee\ P[aux[d]] \leq \text{máx}(P[i], d)\ \}$
si $P[i] > d$ **entonces** $\{$ puede hacerse $\}$
$\{$ desplazar un día los Pokémon posteriores $\}$
para $j = k$ **hasta** $d + 1$ **paso** -1 **hacer** $aux[j+1] := aux[j]$ **fpara**
$aux[d+1] := i\ ;\ k := k + 1$
fsi
fpara
$Pokémon[1..k] := aux[1..k]$
ffun

Coste: $\Theta(n^2)$

Comprobación de la factibilidad, segunda posibilidad

Lema 2

Un conjunto P de Pokémon es factible si y solo si la secuencia de los Pokémon de P donde estos se capturan lo más tarde posible, es admisible.

Demostración:

lo más tarde posible = el día libre más tardío y que no se pase de plazo:

$$d(i) = \text{máx}\{\, d \mid 1 \leq d \leq \text{mín}\{n, p_i\} \ \wedge\ (\forall j : 1 \leq j < i : d \neq d(j))\,\}.$$

($\Leftarrow$) Trivial.
($\Rightarrow$) Por contrarrecíproco.
Supongamos que existe algún Pokémon en P tal que todos los días antes de que expire su plazo p están ocupados.

Para $l = \text{mín}\{n, p\}$ sea $r > l$ el primer día libre; entonces en P hay al menos r Pokémon con plazo menor que r, y por tanto es imposible capturar todos los Pokémon de P dentro de su plazo.

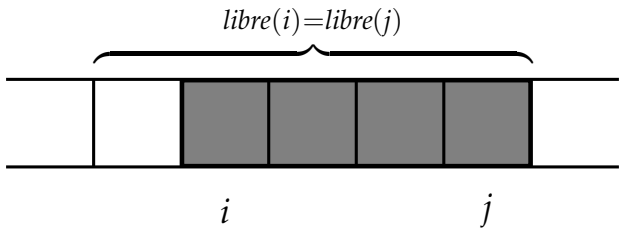
El conjunto P no es factible.

Implementación del test de factibilidad

Definimos una relación de equivalencia sobre los días de captura.

$libre(i) = \text{máx}\{\, d \leq i \mid d \text{ libre} \,\}$, es decir, el primer predecesor libre de i.

i y j estarán en la misma clase si y solo si $libre(i) = libre(j)$.



Para que $libre(i)$ esté definido incluso si todos los días están ocupados, se considera también el día 0, que permanecerá siempre libre.

Cada Pokémon P_i debería capturarse en el día $libre(p_i)$, porque representa el último día libre que respeta su plazo.

Implementación del test de factibilidad

$libre(i)$ cambia a medida que se va planificando la captura de los Pokémon.

Inicio No se ha planificado ninguna captura y cada día está en una clase de equivalencia diferente:

$$\forall i : 1 \leq i \leq n : libre(i) = i.$$

Captura de P_i en $libre(p_i) \neq 0$ Hay que fusionar la clase de equivalencia a la que pertenece $libre(p_i)$ con la correspondiente al día anterior.

Utilizar una estructura de partición en conjuntos disjuntos.

$\{\ V[1] \geq V[2] \geq \ldots \geq V[n]\ \}$

fun `Pokémon-v2`$(P[1..n], V[1..n]$ **de** $nat^+)$ **dev** $\langle\, Pokémon[1..n]$ **de** $1..n, k : 1..n \,\rangle$

var $L[0..n], aux[1..n]$ **de** $0..n, partición$: *partición*$[0..n]$

$l := P[1]$

para $i = 2$ **hasta** n **hacer**

$l :=$ `máx`$(l, P[i])$

fpara

$l :=$ `mín`(n, l)

$partición :=$ `crear-partición`$(l+1)$ $\{\ \{0\}, \{1\}, \ldots, \{l\}\ \}$

$aux[1..l] := [0]$

para $i = 0$ **hasta** l **hacer**

$L[i] := i$

fpara

para $i = 1$ **hasta** n **hacer** $\{$ recorrer Pokémon de mayor a menor valor $\}$

$c_1 :=$ `buscar`$($`mín`$(n, P[i]), partición)$

$m := L[c_1]$

si $m \neq 0$ **entonces**

$aux[m] := i$

$c_2 :=$ `buscar`$(m-1, partición)$

`unir`$(c_1, c_2, partición)$; $L[c_1] := L[c_2]$

fsi

fpara

```
    { comprimir solución }
    k := 0
    para i = 1 hasta l hacer
        si aux[i] > 0 entonces
            k := k + 1 ;  aux[k] := aux[i]
        fsi
    fpara
    Pokémon[1..k] := aux[1..k]
ffun
```

Coste: $\Theta(n \log n)$

Árboles de recubrimiento de coste mínimo

Los residentes de *Barro City* son demasiado tacaños para pavimentar las calles de la ciudad; después de todo, a nadie le gusta pagar impuestos. Sin embargo, tras varios meses de lluvias intensas empiezan a estar cansados de enfangarse los pies cada vez que salen a la calle.

Debido a su gran tacañería, en vez de pavimentar todas las calles de la ciudad, quieren pavimentar solamente suficientes calles para poder ir de una intersección a otra en la ciudad siguiendo una ruta pavimentada, y sobre todo quieren gastarse tan poco dinero como sea posible en la realización de esta pavimentación.

A los residentes de *Barro City* no les importa caminar mucho con tal de ahorrar dinero.

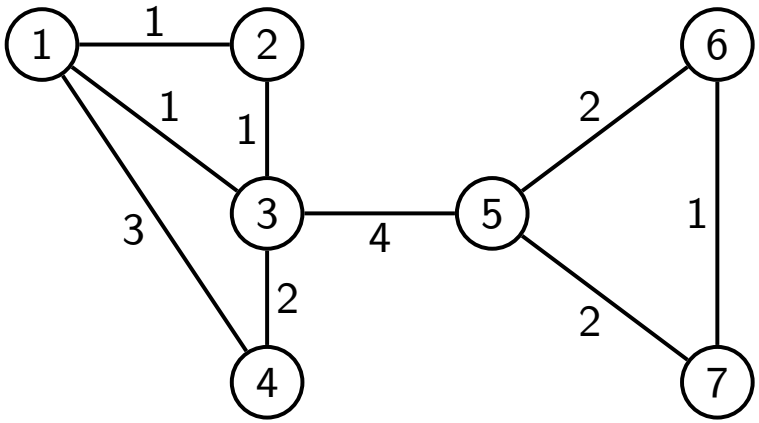
Árboles de recubrimiento de coste mínimo

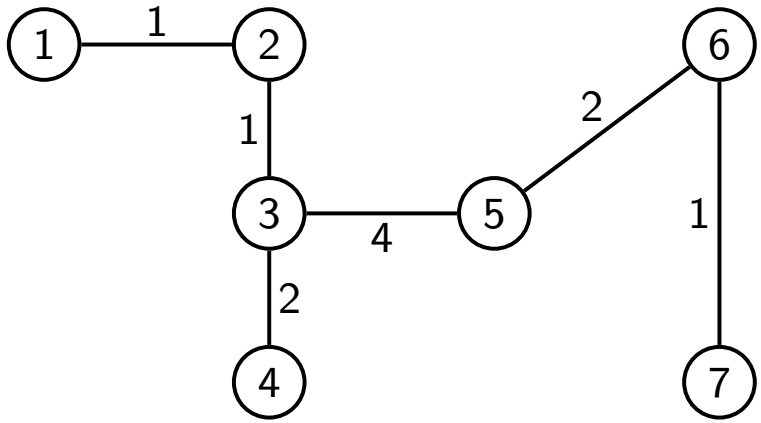
$G = \langle N, A \rangle$ grafo conexo, no dirigido y valorado.

- Hallar $T \subseteq A$ tal que $G' = \langle N, T \rangle$ siga siendo conexo y la suma de las aristas de T sea lo *menor* posible.
- Un grafo conexo con n vértices debe tener al menos $n - 1$ aristas.
- Un grafo con n vértices y más de $n - 1$ aristas contiene al menos un ciclo.

T debe tener $|N| - 1$ aristas y formar un árbol.

G' se denomina árbol de recubrimiento de coste mínimo (ARM).





ARM: Algoritmos voraces

Mirar todas las posibilidades requiere un tiempo exponencial en el caso peor.

Algoritmo voraz:

$AR := \emptyset$; $candidatas := A$
mientras queden vértices sin conectar **hacer**
    $a :=$ `seleccionar`$(candidatas)$;
    $candidatas := candidatas - \{a\}$
    si `sin-ciclos?`$(AR \cup \{a\})$ **entonces** $AR := AR \cup \{a\}$ **fsi**
fmientras

Dos estrategias:

Algoritmo de Kruskal Seleccionar en cada etapa la arista más corta.

Algoritmo de Prim A partir de un vértice cualquiera, construir un árbol seleccionando en cada etapa la arista más corta que extienda el árbol.

ARM: Propiedad

- Un conjunto de aristas es prometedor si puede extenderse hasta un ARM.
- Una arista sale de un conjunto de vértices si tiene exactamente un extremo en el conjunto.

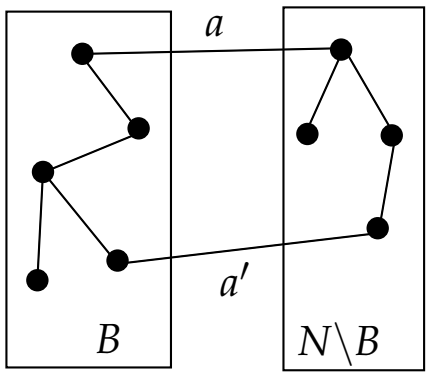
Lema

$G = \langle N, A \rangle$ grafo conexo, no dirigido y valorado.

- subconjunto $B \subset N$,
- $T \subseteq A$ conjunto prometedor tal que ninguna arista de T sale de B,
- a la arista más corta que sale de B.

Entonces $T \cup \{a\}$ es prometedor.

Demostración: Sea $\mathcal{U}$ un ARM de G tal que $T \subseteq \mathcal{U}$.
Si $a \in \mathcal{U}$, $T \cup \{a\}$ es prometedor. Si no, añadiendo a a $\mathcal{U}$ creamos un ciclo.



Eliminando a' el ciclo desaparece:
$\mathcal{V} = \mathcal{U} \cup \{a\} - \{a'\}$ es un árbol que cubre a G.
La longitud de $\mathcal{V}$ no sobrepasa la de $\mathcal{U}$
$\Rightarrow$ $\mathcal{V}$ también es ARM, y $T \subseteq \mathcal{V}$
$\Rightarrow$ $T \cup \{a\}$ es prometedor.

ARM: Algoritmo de Kruskal

Inicio T vacío y cada vértice pertenece a una componente conexa distinta.

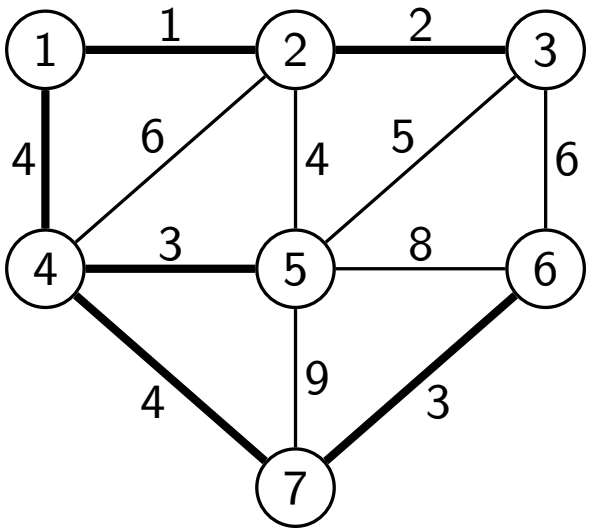
Propiedad Las aristas de T pertenecientes a cada componente conexa forman un ARM para dicha componente.

Selección Para hacer crecer T, se examinan las aristas por orden creciente de longitudes.

Factibilidad Si la arista seleccionada une dos vértices de la misma componente se rechaza, y si une dos vértices de componentes distintas, se añade a T.

Final Solo hay una componente conexa $\Rightarrow$ T es un ARM para G.

Algoritmo de Kruskal: Ejemplo



Paso	Arista	Componentes conexas
Inic.	-	{1} {2} {3} {4} {5} {6} {7}
1	$\langle 1,2 \rangle$	{1,2} {3} {4} {5} {6} {7}
2	$\langle 2,3 \rangle$	{1,2,3} {4} {5} {6} {7}
3	$\langle 4,5 \rangle$	{1,2,3} {4,5} {6} {7}
4	$\langle 6,7 \rangle$	{1,2,3} {4,5} {6,7}
5	$\langle 1,4 \rangle$	{1,2,3,4,5} {6,7}
6	$\langle 2,5 \rangle$	rechazada
7	$\langle 4,7 \rangle$	{1,2,3,4,5,6,7}

Algoritmo de Kruskal: Corrección

Proposición 1

Dado un grafo $G = \langle N, A \rangle$, el algoritmo de Kruskal obtiene un ARM de G.

Demostración: T siempre es prometedor.

Inducción sobre el número de aristas de T:

Base: $T = \emptyset$ es prometedor, porque G es conexo.

Hipótesis de inducción: T es prometedor antes de añadir la arista $a = \langle u, v \rangle$.

Paso inductivo: Sea B el conjunto de vértices de la componente conexa que contiene a u:

- $B \subset N$,
- T es prometedor y ninguna arista de T sale de B,
- a es una de las aristas más cortas que sale de B.

Por el lema anterior, $T \cup \{a\}$ es prometedor.

Algoritmo de Kruskal: Implementación

```
{ G es conexo }
fun Kruskal(G : grafo-val[n]) dev ARM : conjunto[arista-val]
var P : partición[1..n], A[1..n^2] de arista-val, a : arista-val
    ARM := cjto-vacío() ; núm-aristas := 0
    P := crear-partición(n)
    ⟨A, m⟩ := obtener-vector-aristas(G)    { aristas en A[1..m] }
    ordenar(A, m)    { por costes crecientes }
    i := 1
    mientras núm-aristas < n − 1 hacer
        a := A[i] ; i := i + 1    { seleccionar arista siguiente }
        c := buscar(a.origen, P) ; d := buscar(a.destino, P)
        si c ≠ d entonces
            unir(c, d, P)
            añadir(a, ARM) ; núm-aristas := núm-aristas + 1
        fsi
    fmientras
ffun
```

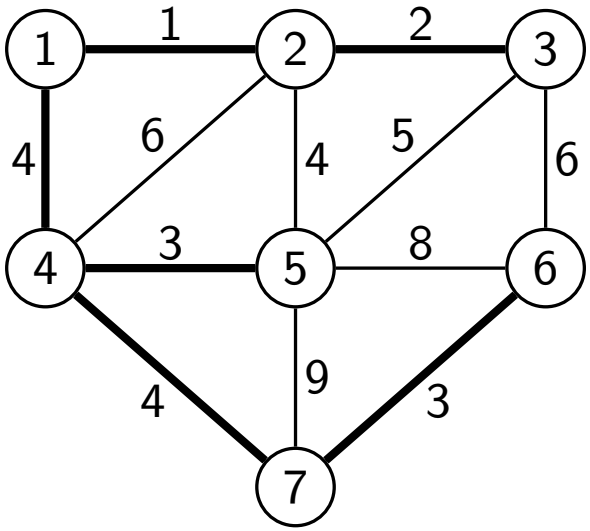
Coste: en $\Theta(m \log n)$ para grafos implementados mediante listas de adyacentes; en $\Theta(n^2 + m \log n)$ para grafos con matriz de valores.

ARM: Algoritmo de Prim

- El ARM va creciendo empezando por una raíz arbitraria.
- En cada etapa se añade una nueva rama al árbol.
- El algoritmo termina cuando el árbol llega a todos los vértices.

Inicio	B contiene un único vértice y T está vacío.
Selección	Seleccionar la arista más corta $\langle u, v \rangle$ que sale de B ($u \in B$ y $v \notin B$). Añadir v a B y $\langle u, v \rangle$ a T.
Propiedad	T es un ARM para los vértices de B.
Final	$B = N \Rightarrow T$ es un ARM para G.

Algoritmo de Prim: Ejemplo



Paso	$\langle u, v \rangle$	B
Inic.	-	$\{1\}$
1	$\langle 1,2 \rangle$	$\{1,2\}$
2	$\langle 2,3 \rangle$	$\{1,2,3\}$
3	$\langle 1,4 \rangle$	$\{1,2,3,4\}$
4	$\langle 4,5 \rangle$	$\{1,2,3,4,5\}$
5	$\langle 4,7 \rangle$	$\{1,2,3,4,5,7\}$
6	$\langle 7,6 \rangle$	$\{1,2,3,4,5,6,7\}$

Algoritmo de Prim: Corrección

Proposición 2

Dado un grafo $G = \langle N, A \rangle$, el algoritmo de Prim obtiene un ARM de G.

Demostración: T siempre es prometedor.

Inducción sobre el número de aristas de T:

Base: $T = \emptyset$ es prometedor, porque G es conexo.

Hipótesis de inducción: T es prometedor antes de añadir la arista $a = \langle u, v \rangle$.

Paso inductivo:

- $B \subset N$,
- T es prometedor,
- a es una de las aristas más cortas que sale de B.

Por el lema anterior, $T \cup \{a\}$ es prometedor.

Algoritmo de Prim: Implementación

Representación del grafo mediante su matriz de costes:

$$G[i,j] = \begin{cases} \textit{coste} & \text{si } \langle i,j \rangle \in A \\ +\infty & \text{si } \langle i,j \rangle \notin A \end{cases}$$

- Para los vértices $i \notin B$

  $\textit{conexión}[i]$ vértice de B más próximo a i,
  $\textit{dist-mín}[i]$ distancia de i a este vértice.
- Para los vértices $i \in B$, $\textit{dist-mín}[i] = -1$.

Vértice inicial: 1

Algoritmo de Prim: Implementación

```
fun Prim(G : grafo-val[n]) dev ARM : conjunto[arista]
var dist-mín[2..n] de real_∞, conexión[2..n] de 1..n, a : arista
    ARM := cjto-vacío()
    para i = 2 hasta n hacer
        dist-mín[i] := G[1,i] ; conexión[i] := 1
    fpara
    para i = 1 hasta n − 1 hacer
        { encontrar el mínimo en dist-mín }
        mínimo := +∞
        para j = 2 hasta n hacer
            si 0 ≤ dist-mín[j] ∧ dist-mín[j] < mínimo entonces
                mínimo := dist-mín[j] ; elegido := j
            fsi
        fpara
        a.origen := elegido ; a.destino := conexión[elegido] ; añadir(a, ARM)
        dist-mín[elegido] := −1
        para j = 2 hasta n hacer { actualizar los vectores }
            si G[elegido, j] < dist-mín[j] entonces
                dist-mín[j] := G[elegido, j]
                conexión[j] := elegido
            fsi
        fpara
    fpara
ffun
```

Coste: $\Theta(n^2)$ en tiempo, y $\Theta(n)$ en espacio

ARM: Comparación

Grafo denso $m \in O(n^2)$

Kruskal $O(n^2 \log n)$

Prim $O(n^2)$

Grafo disperso $m \in O(n)$

Kruskal $O(n \log n)$

Prim $O(n^2)$

Utilizando montículos se puede obtener un coste para Prim en $\Theta(m \log n)$ en tiempo, y $\Theta(n)$ en espacio:

Grafo denso $O(n^2 \log n)$

Grafo disperso $O(n \log n)$

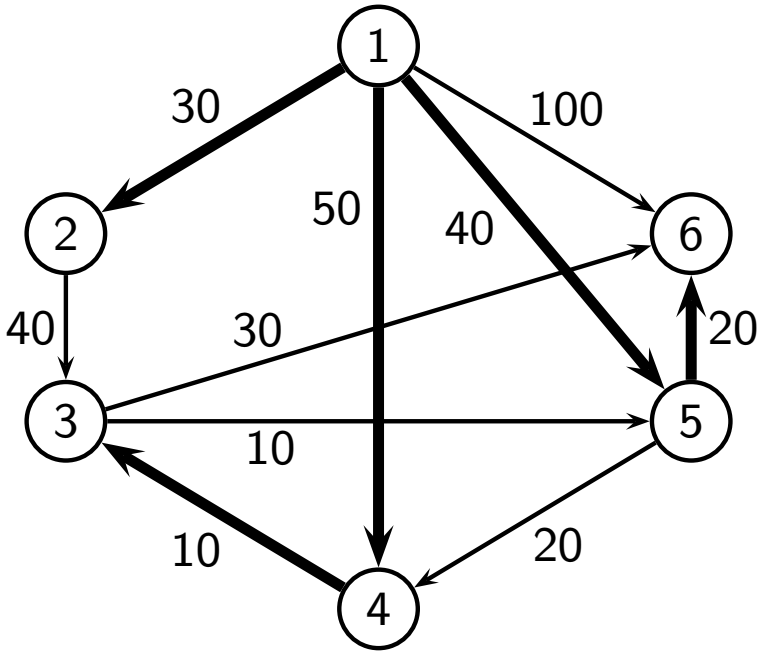
Caminos mínimos

El fantasma de la zarzuela vive oculto en las salas más recónditas del viejo Teatro de la Zarzuela de *Barro City*. El fantasma conoce la existencia de una red de túneles que recorren el subsuelo de la ciudad, y lo utiliza para desplazarse en sus correrías nocturnas. Por los túneles transcurre una corriente subterránea, de forma que hay que desplazarse por ellos mediante una barca que es arrastrada en el sentido de la corriente. La red tiene una entrada secreta desde el teatro y diversas salidas en puntos estratégicos de la ciudad.
El fantasma desea conocer el camino más corto para desplazarse por la red subterránea desde el teatro a los otros puntos estratégicos.

Grafo $G = \langle N, A \rangle$ dirigido y valorado,
$N = \{1, 2, \ldots, n\}$, aristas de longitud positiva:

caminos mínimos desde el origen hasta cada uno de todos los demás vértices.

Caminos mínimos: Ejemplo



Caminos mínimos: Algoritmo de Dijkstra

Vértices N se particiona en dos conjuntos:

- $S =$ vértices seleccionados; la distancia mínima desde el origen ya es conocida;
- $C =$ los candidatos restantes.

Inicio S solo contiene el origen.

Final $S = N$.

Selección En cada etapa, seleccionar un vértice de C cuya distancia al origen sea mínima.

Propiedad El vector D contiene la longitud del camino interno más corto desde el origen a cualquier otro vértice.
Camino interno: desde el origen a cualquier otro vértice cuyos vértices intermedios pertenecen a S.
Para los vértices en S, D contiene la distancia mínima desde el origen.

Obtener caminos (no solo la longitud) Utilizar un vector P con el predecesor de cada vértice en el camino interno mínimo.
Recorrer los predecesores hacia atrás hasta alcanzar el origen.

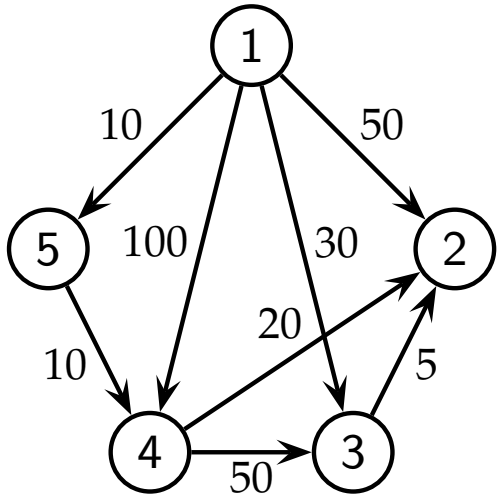
Caminos mínimos: Algoritmo de Dijkstra

Matriz de costes $G[i,j] = \begin{cases} \textit{coste} & \text{si } i \to j \in A \\ +\infty & \text{si } i \to j \notin A \end{cases}$

```
fun Dijkstra(G[1..n,1..n]) dev ⟨ D[2..n], P[2..n] ⟩      { O(n^2) }
    C := {2,3,...,n}
    para i = 2 hasta n hacer
        D[i] := G[1,i]
        P[i] := 1
    fpara
    repetir n − 2 veces
        v := vértice en C tal que D[v] sea mínimo
        C := C − {v}
        para cada w ∈ C hacer
            si D[v] + G[v,w] < D[w] entonces
                D[w] := D[v] + G[v,w]
                P[w] := v
            fsi
        fpara
    frepetir
ffun
```

Versión más eficiente ($O(m \log n)$, n vértices y m aristas) con montículos.

Algoritmo de Dijkstra: Ejemplo



Paso	v	C	D	P
inic.	–	$\{2,3,4,5\}$	$[50,30,100,\underline{10}]$	$[1,1,1,1]$
1	5	$\{2,3,4\}$	$[50,30,\underline{20},10]$	$[1,1,5,1]$
2	4	$\{2,3\}$	$[40,\underline{30},20,10]$	$[4,1,5,1]$
3	3	$\{2\}$	$[\underline{35},30,20,10]$	$[3,1,5,1]$
4	2	$\{\}$	$[35,30,20,10]$	$[3,1,5,1]$

Algoritmo de Dijkstra: Corrección

Proposición 3

El algoritmo de Dijkstra obtiene los caminos más cortos desde el origen (v_1) hasta los demás vértices.

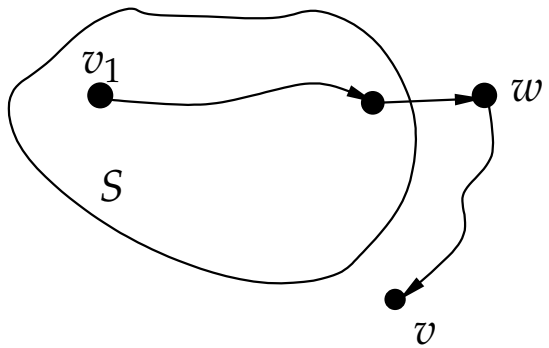
Demostración: Por inducción sobre el número de pasos del algoritmo.

(a) $v_i (\neq v_1) \in S \Rightarrow D[i] =$ longitud del camino más corto desde v_1 a v_i.
(b) $v_i \notin S \Rightarrow D[i] =$ longitud del camino *interno* más corto desde v_1 a v_i.

Base: $S = \{v_1\} \Rightarrow$ (a) se cumple trivialmente y (b) por cómo se inicializa D.

Paso inductivo (a): Comprobar que tras añadir v sigue cumpliéndose (a).

- Para los vértices que ya estaban, no cambia nada.
- No hay otro camino interno a v de longitud menor.



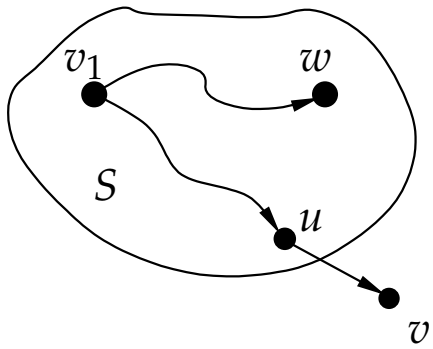
$$d(v_1, v) = d(v_1, w) + d(w, v) = D[w] + d(w, v) \geq D[v]$$

Algoritmo de Dijkstra: Corrección

Paso inductivo (b): Comprobar que tras añadir v sigue cumpliéndose (b).

$$\forall w \in S : d(v_1, w) \leq d(v_1, v)$$

Se demuestra por inducción sobre el número de etapas:
En la primera etapa es trivial. Sea la etapa en que se selecciona v y sea u el predecesor de v en el camino mínimo desde v_1 a v:

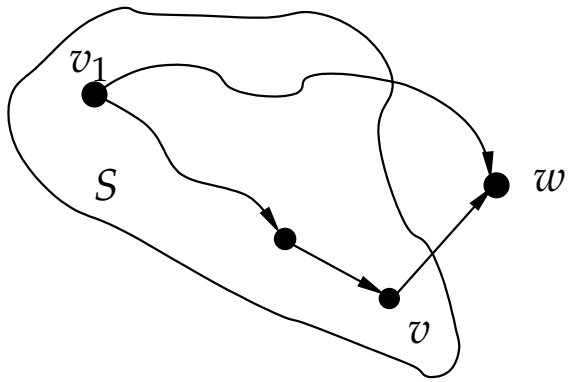


- u se incorporó a S en una etapa anterior que w:
  al seleccionar w, v ya era accesible pero no fue seleccionado
  $\Rightarrow d(v_1, w) = D[w] \leq D[v] = d(v_1, v)$.
- u se incorporó a S en una etapa posterior que w:
  por h.i. $d(v_1, w) \leq d(v_1, u)$, y $d(v_1, u) \leq d(v_1, v)$.

Algoritmo de Dijkstra: Corrección

Al añadir v a S:

- el camino interno más corto a w no cambia, o bien
- el camino interno más corto a w pasa por v.
  - v es el último vértice en este camino interno más corto a w:



El algoritmo compara $D[w]$ con $D[v] + G[v,w]$ y se queda con el menor.

- v es un vértice intermedio en el camino interno más corto a w y u el último vértice interno, entonces $d(v_1,u) \leq d(v_1,v)$
  $\Rightarrow$ el nuevo camino es al menos tan largo como ir desde v_1 a u y de ahí a w.
  $\Rightarrow$ No es necesario considerar ese tipo de caminos.


“Greed … is good. Greed is right.
Greed works.”