

Programación y Estructuras de Datos Avanzadas

Capítulo 3: Algoritmos voraces

Capítulo 3: Algoritmos voraces

3.1 Planteamiento y esquema general

- El **esquema voraz** (*greedy algorithms*) se aplica a problemas de optimización en los que la solución se puede construir paso a paso sin necesidad de reconsiderar decisiones ya tomadas.
 - El problema se interpreta como: “*seleccionar algunos elementos de entre un conjunto de n candidatos que constituya una solución y que optimice (maximice o minimice) una función objetivo*”.
 - El **orden** el que se eligen estos elementos puede ser importante o no.
- Un **algoritmo voraz** funciona por pasos:
 - Inicialmente partimos de una solución vacía.
 - En cada paso se escoge *el candidato más prometedor de los disponibles* (relacionado con la función objetivo) y *se decide si se incluye o no en la solución*.
 - Una vez tomada **esta decisión no se podrá deshacer**.
 - El algoritmo acabará cuando el conjunto de elementos seleccionados constituya una solución.

• Esquema general de un Algoritmo voraz:

Entrada: c conjunto de candidatos inicial Salida: conjunto solución (subconjunto de c)

```

fun Voraz( $c$ : conjuntoCandidatos): conjuntoCandidatos
    sol  $\leftarrow \emptyset$       1. Comenzamos con un conjunto solución vacío
    mientras  $c \neq \emptyset \wedge \neg \text{solucion}(\text{sol})$  hacer      Itera 2 a 5 mientras queden candidatos y no se haya alcanzado la solución
         $x \leftarrow \text{seleccionar}(c)$       2. Función de selección: devuelve el candidato más prometedor de los que todavía no se han considerado. Relacionado con la optimización. Necesaria demostración de optimalidad
         $c \leftarrow c \setminus \{x\}$       3. Se elimina dicho candidato del conjunto de candidatos posibles ( $c$ )
        si  $\text{factible}(\text{sol} \cup \{x\})$  entonces      4. Se comprueba si el conjunto resultante de incorporar el candidato a la solución es factible o completable (es decir si cumple las restricciones al problema para llegar a ser solución)
             $\text{sol} \leftarrow \text{sol} \cup \{x\}$ 
        fsi
    fmientras
    si  $\text{solucion}(\text{sol})$  entonces devolver  $\text{sol}$       Determina si el conjunto factible sol es solución o no
    sino imprimir('no hay solución')
    fsi
ffun

```

Características:

- Al no reconsiderar decisiones son algoritmos eficientes.
- Aunque puedan parecer sencillos → demostración optimalidad compleja.

• Ejemplo: Problema del cambio de monedas

- Contamos con un conjunto de monedas de determinados valores.
- Problema: *construir un algoritmo que dada una cantidad P devuelva esa cantidad usando el menor número posible de monedas.*

Disponemos un nº ilimitado de monedas con valores de 1, 2, 5, 10, 20 y 50 céntimos de euro, 1 y 2 euros (€).



Caso 1. Devolver 3,89 Euros.

1 moneda de 2€, 1 moneda de 1€, 1 moneda de 50c, 1 moneda de 20c, 1 moneda de 10c , 1 moneda de 5c y 2 monedas de 2c.

Total: 8 monedas



El método intuitivo se puede entender como un **algoritmo voraz**: en cada paso añadir una moneda nueva a la solución actual, hasta llegar a P.

Adaptación de este problema al esquema general:

- **Conjunto de candidatos:** todos los tipos de monedas disponibles. Supondremos una *cantidad ilimitada* de monedas de cada tipo.
- **Solución:** conjunto de monedas que sumen P .
- **Función objetivo:** minimizar el número de monedas.

Funciones del algoritmo:

- **Seleccionar:** escoge la moneda de mayor valor que todavía no ha sido descartada. *Habrá que demostrar su optimalidad.*
 - **Factible:** comprueba si el valor de las monedas seleccionadas en s es $\leq P$.
 - **Solucion:** comprueba si el valor de las monedas de s es igual a P .
- *En la práctica , en vez de ir de 1 en 1 moneda, seleccionar utiliza el cociente de la división entera: $P \text{ div } m$ (siendo m la siguiente moneda de mayor valor)=nº de monedas de valor m que hay incluir en la solución. De esa forma factible es siempre cierto.*

Optimalidad: ¿está asegurada la solución óptima?

- *Depende del sistema monetario.*
- *Ejemplo. Supongamos que tenemos monedas de 1€, 90c y 1c y queremos devolver 1,80 € ($P=1,80\text{€}$).*

- **Algoritmo voraz.** 1 moneda de 1€ y 80 monedas de 1c: total 81 monedas.
- **Solución óptima.** 2 monedas de 90c: total 2 monedas → ***no sirve voraz.***



Consideraciones:

- Vamos a suponer que tenemos *n tipos de moneda* que son *potencias consecutivas de m*.
- Tipos de moneda:** $T=\{m^0, m^1, m^2, \dots, m^{n-1}\}$
- **Conjunto de candidatos:** una cantidad ilimitada de cada tipo de moneda de T.

tipo VectorNat = matriz[1..n] de natural

fun MonedasCambio(T: VectorNat, C: natural): VectorNat

var

solucion: VectorNat

fvar

para i $\leftarrow 1$ **hasta** n **hacer**

solucion[i] $\leftarrow 0$

fpara

canRestante $\leftarrow C$

i $\leftarrow n$ Comienzo con el último elemento del vector $T[i]$,
o lo que es lo mismo **con la moneda más alta**

mientras canRestante $\neq 0 \wedge i \geq 1$ **hacer**

solucion[i] \leftarrow canRestante **div** T[i] Asigna las siguientes monedas (**no necesita**
factible, porque si no caben ya asigna 0)

canRestante \leftarrow canRestante **mod** T[i] Actualiza la cantidad restante

i $\leftarrow i - 1$ Pasa al siguiente tipo de monedas

fmientras

dev solucion[]

ffun

Podemos identificar los elementos del esquema voraz en este algoritmo:

- Resolución de un problema de forma óptima: descomponer una cantidad en un número mínimo de monedas.
- *Conjunto inicial de candidatos*: los distintos tipos de moneda.
- *Conjunto de candidatos seleccionados y conjunto de candidatos rechazados*: están representados en el array *solucion[]* desde las posiciones n-ésima a la i-ésima. Los candidatos seleccionados tendrán en su posición del array un valor mayor que 0, y los rechazados tendrán un valor igual a 0.
- Las funciones *solución* y *factible* están implícitas en las condiciones de finalización del bucle.
- La función de selección se traduce en la selección de los elementos del array $T[]$ según el orden: $n, n-1, n-2, \dots, 1$.
- La función objetivo no aparece explícitamente en el algoritmo.

Coste: se ejecutan 2 bucles consecutivos de n iteraciones con todas las instrucciones elementales
 $\rightarrow O(n)$, siendo n el nº de tipos de monedas disponibles.

Demostración de optimalidad

- Los algoritmos voraces son sencillos y rápidos pero requieren la demostración de optimalidad
→ Demostrar que la función de selección conduce a la solución óptima: no suele ser sencillo.

La demostración de optimalidad se apoya en la siguiente propiedad general de los números naturales: si m es un número natural mayor que 1, todo número natural c puede expresarse de forma única como:

$$c = v_0m^0 + v_1m^1 + v_2m^2 + \cdots + v_nm^n \quad (3.1)$$

siendo $0 \leq v_i < m$ para todo $0 \leq i \leq n$ y siendo n el menor natural tal que $c < m^{n+1}$. Como el algoritmo lo que hace es calcular los v_i asociados a los m_i , es decir, el número de monedas de cada tipo en que se descompone la cantidad c , hay que demostrar que la descomposición que se obtiene con los v_i es óptima. Esto supone que si:

$$c = s_0m^0 + s_1m^1 + s_2m^2 + \cdots + s_pm^p$$

es una descomposición distinta de la misma cantidad c , entonces:

$$\sum_{i=0}^n v_i < \sum_{i=0}^p s_i$$

La demostración se hará para el caso más sencillo $m = 2$ ya que el caso general es similar. Sea la descomposición obtenida por el algoritmo voraz la siguiente:

$$c = v_02^0 + v_12^1 + v_22^2 + \cdots + v_n2^n = v_0 + 2v_1 + 2^2v_2 + \cdots + 2^nv_n \quad (3.2)$$

tenemos que $c < 2^{n+1}$ y que $0 \leq v_i < 2$ por lo que los coeficientes v_i tendrán el valor 0 o 1. Si la siguiente descomposición es distinta:

$$c = s_0 + 2s_1 + 2^2s_2 + \cdots + 2^ps_p$$

como tenemos que $c < 2^{n+1}$, implica que $p \leq n$ (porque si $p > n$ entonces se daría que $m^p > c$). Con el fin de tener n términos en cada descomposición se definen $s_{p+1} = s_{p+2} = \cdots = s_n = 0$. Se trata entonces de demostrar que:

$$v_0 + v_1 + v_2 + \cdots + v_n < s_0 + s_1 + s_2 + \cdots + s_n$$

Como ambas descomposiciones son distintas, sea k el primer índice tal que $v_k \neq s_k$. Supongamos sin pérdida de generalidad que $k = 0$ (en caso de que $k \neq 0$ se podrían eliminar de la desigualdad los términos iguales y dividir por la potencia de 2 correspondiente), como $v_0 \neq s_0$ veamos cómo es v_0 con respecto a s_0 .

- Si c es par, entonces $v_0 = 0$. Como $s_0 \geq 0$ y $v_0 \neq s_0$ entonces $v_0 < s_0$.
- Si c es impar, entonces $v_0 = 1$ y por lo tanto $s_0 \geq 1$. Como también sabemos que $v_0 \neq s_0$, entonces $s_0 > 1$ y en este caso también $v_0 < s_0$.

Por lo anterior, $s_0 - v_0 > 0$. Además, esta cantidad debe ser par ya que siendo $m = 2$ la cantidad $c - v_0$ es par por la expresión 3.2. Al ser par, siempre se podrá mejorar la descomposición (s_0, s_1, \dots, s_n) cambiando $s_0 - v_0$ monedas de 1 unidad por $(s_0 - v_0)/2$ monedas de 2 unidades, obteniendo así:

$$s_0 + s_1 + s_2 + \cdots + s_n > v_0 + (s_1 + \frac{s_0 - v_0}{2}) + s_2 + \cdots + s_n \quad (3.3)$$

Utilizando el razonamiento anterior se ha obtenido una nueva descomposición mejor y manteniendo:

$$v_0 + (s_1 + \frac{s_0 - v_0}{2})2 + s_22^2 + \cdots + s_n2^n = c = v_0 + v_12 + \cdots + v_n2^n$$

Si aplicamos sucesivamente el razonamiento anterior a la nueva descomposición se puede ir viendo que $s_i \geq v_i$ para todo $0 \leq i \leq n-1$ y así ir obteniendo nuevas descomposiciones, cada una mejor que la anterior, hasta llegar en el último paso a la siguiente descomposición:

$$v_0 + v_12 + v_22^2 + \cdots + v_{n-1}2^{n-1} + (s_n + \frac{s_{n-1} - acum_{n-1}}{2})2^n = c \quad (3.4)$$

en la que se han ido acumulando las diferencias en el último término y que además verifica que:

$$v_0 + v_1 + \cdots + v_i + (s_{i+1} + \frac{s_i - acum_i}{2}) + \cdots + s_n \geq v_0 + v_1 + \cdots + v_n, \quad 0 \leq i \leq n-1$$

Si se verifica 3.4 por la unicidad de la descomposición a la que hacía referencia la propiedad 3.1 se ha de cumplir que:

$$s_n + \frac{s_{n-1} - acum_{n-1}}{2} = v_n$$

que junto a las desigualdades 3.3 hace que quede demostrada la afirmación. El razonamiento es igual para $m > 2$.

3.2 Algoritmos voraces con grafos

- **Árbol de recubrimiento mínimo (ARM):**
 - Sea $G = \langle N, A \rangle$ grafo conexo y no dirigido cuyas aristas tienen asignados pesos no negativos.
 - **Objetivo:** cálculo del subconjunto ARM de aristas de G de forma que conecte todos sus nodos con un peso total mínimo. Contiene exactamente $n-1$ aristas y puede no ser único si hay aristas de igual peso.
 - Dos algoritmos voraces para resolver este problema: **algoritmo de Prim** y **algoritmo de Kruskal**.
 - Suponiendo que los nodos son ciudades y las aristas son carreteras de una determinada longitud/coste: encontrar que carreteras tengo que construir para que estén conectadas todas las ciudades y con un coste total mínimo. También se utiliza para asegurar la conectividad en redes, etc.
- **Camino de coste mínimo:**
 - Sea $G = \langle N, A \rangle$ grafo dirigido, con pesos mayores o iguales que cero en sus aristas y en el que los nodos son accesibles desde uno concreto considerado como origen.
 - **Objetivo:** determinar la longitud del camino de coste mínimo que va desde el nodo origen a cada uno de los demás nodos del grafo → **algoritmo de Dijkstra**.
 - Suponiendo que los nodos son ciudades y las aristas carreteras esto permitiría calcular la ruta más corta entre una ciudad y todas las demás o entre 2 ciudades cualesquiera.

3.2.1 Árboles de recubrimiento mínimo: algoritmo de Prim

- Sea $G = \langle N, A \rangle$ grafo conexo y no dirigido cuyas aristas tienen asignados pesos no negativos.
 - i. Selecciona un nodo del grafo como raíz del árbol de recubrimiento (AR).
 - ii. En cada paso, se añade al árbol una arista de coste mínimo (u, v) tal que $AR \cup \{(u, v)\}$ sea también un árbol (esto implica que uno de los nodos u o v ya estaban en AR pero el otro no).
 - iii. El algoritmo termina hasta que AR contiene $n-1$ aristas.

fun Prim ($G = \langle N, A \rangle$: grafo): conjunto de aristas

$AR \leftarrow \emptyset$

 ← Inicializo el conjunto de aristas del árbol de recubrimiento mínimo

$NA \leftarrow \{ \text{un nodo cualquiera de } N \}$

 ← NA: conjunto de nodos en el AR. Lo inicializo tomando cualquier nodo como raíz.

mientras $NA \neq N$ **hacer**

 Buscar $\{u, v\}$ de coste mínimo tal que $u \in NA$ y $v \in N \setminus NA$

$AR \leftarrow AR \cup \{(u, v)\}$
 $NA \leftarrow NA \cup \{v\}$

 En cada paso cojo la arista mas corta que conecte un nodo que ya está en el árbol de recubrimiento (u) con uno nodo de los que aún no están en el árbol (v). **No necesito factible.**

fmientras

dev AR

 Incorporo la arista a la solución y el nodo v al árbol de recubrimiento.

ffun

Demostración de optimalidad algoritmo de Prim

- **Conjunto prometedor:** conjunto de aristas factible (sin ciclos) que se puede extender hasta llegar a la solución óptima. **Arista que sale** de un conjunto de nodos: sólo tiene un extremo en él.

Lema 3.2.1 *Sea $G = \langle N, A \rangle$ un grafo conexo, no dirigido, con pesos mayores o iguales que cero en sus aristas. Sea $NA \subset N$ un subconjunto estricto de los nodos de G . Sea $AR \subseteq A$ un conjunto prometedor de aristas tal que no haya ninguna arista de AR que sale de algún nodo de NA . Sea (u, v) la arista de peso menor que salga de NA (o una de las de peso menor si hay más de una con igual peso), entonces $AR \cup \{(u, v)\}$ es un conjunto prometedor.*

Veamos la demostración de este lema. Sea ARM un árbol de recubrimiento mínimo de G tal que $AR \subseteq ARM$. Nótese que ARM tiene que existir ya que AR es prometedor por hipótesis. Si la arista $(u, v) \in ARM$ no hay nada que demostrar. Si no, al añadir (u, v) a ARM se crea un ciclo. En este ciclo, como (u, v) sale de NA existe necesariamente al menos otra arista (x, y) que también sale de NA o el ciclo no se cerraría. Si eliminamos (x, y) el ciclo desaparece y obtenemos un nuevo árbol ARM' de G . Como la longitud de (u, v) , por definición, no es mayor que la longitud de (x, y) , la longitud total de las aristas de ARM' no sobrepasa la longitud total de las aristas de ARM . Por lo tanto, ARM' es también un árbol de recubrimiento mínimo de G y contiene a (u, v) . Como la arista que se ha eliminado sale de NA , no podría haber sido una arista de AR y se cumple que $AR \subseteq ARM'$.

La demostración de optimalidad que garantiza que el algoritmo de Prim halla un árbol de recubrimiento mínimo se realiza por inducción y parte del lema anterior:

Base: el conjunto vacío es prometedor.

Paso inductivo: suponemos que AR es un conjunto de aristas prometedor antes de que el algoritmo añada una nueva arista (u, v) ; NA es un subconjunto estricto de N ya que el algoritmo termina una vez que $NA = N$; la arista (u, v) es una de las de menor peso de las que salen de NA . Entonces podemos utilizar el lema 3.2.1 ya que se cumplen sus condiciones y por lo tanto $AR \cup \{(u, v)\}$ también es prometedor.

Descripción detallada del algoritmo de Prim

```

tipo VectorNat = matriz[0..n] de natural
tipo VectorEnt = matriz[0..n] de entero
fun PrimDetallado ( $G = \langle N, A \rangle$ : grafo): conjunto de aristas
    var
        nodoMinimo: VectorNat
        costeMinimo: VectorEnt
        AR: conjunto de aristas
    fvar
        AR  $\leftarrow \emptyset$ 
        costeMinimo[1]  $\leftarrow -1$ 
        para i  $\leftarrow 2$  hasta n hacer
            nodoMinimo[i]  $\leftarrow 1$ 
            costeMinimo[i]  $\leftarrow$  Distancia(1,i)
        fpara
        para i  $\leftarrow 1$  hasta n-1 hacer
            min  $\leftarrow \infty$ 
            para j  $\leftarrow 2$  hasta n hacer
                si  $0 \leq \text{costeMinimo}[j] \wedge \text{costeMinimo}[j] < \text{min}$  entonces
                    min  $\leftarrow \text{costeMinimo}[j]$ 
                    nodo  $\leftarrow j$ 
                fsi
            fpara
            AR  $\leftarrow AR \cup \{(nodoMinimo[nodo], nodo)\}$   $\leftarrow$  Añado la arista a la solución
            costeMinimo[nodo]  $\leftarrow -1$ 
            para j  $\leftarrow 2$  hasta n hacer
                si Distancia(j,nodo)  $< \text{costeMinimo}[j] \wedge \text{costeMinimo}[j] \neq -1$  entonces
                    costeMinimo[j]  $\leftarrow$  Distancia(j,nodo)  $\leftarrow$  Función que devuelve la distancia/coste entre 2 nodos
                    nodoMinimo[j]  $\leftarrow$  nodo
                fsi
            fpara
            fpara
            dev AR
        ffun
    
```

- Grafo de n nodos $\rightarrow N=\{1,2,\dots,n\}$ array de n enteros, nodo 1 raíz
- Array **nodoMinimo[i]**, donde el nodo $i \in N \setminus NA \rightarrow$ indica el **nodo de NA más próximo** al nodo i, siendo **costeMinimo[i]** el coste del nodo i a dicho nodo (un nodo que ya esté en NA se indicará asignando su **costeMinimo[i]=-1**).
- En este algoritmo no se utiliza explícitamente NA (conjunto nodos ya incluidos en el árbol).

Ejemplo de ejecución

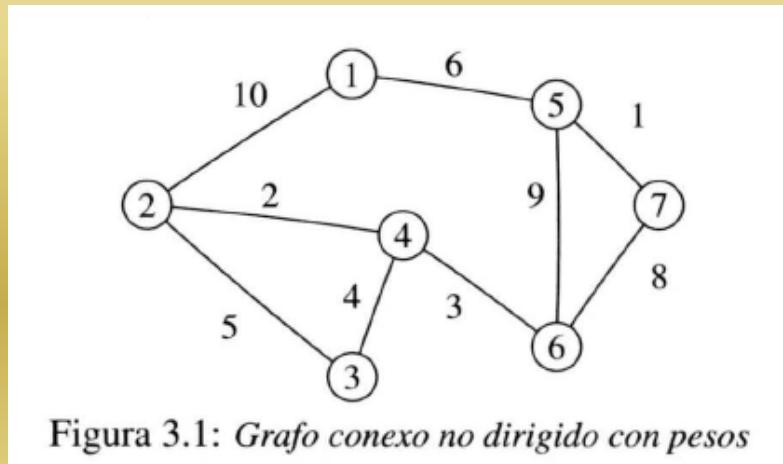


Figura 3.1: Grafo conexo no dirigido con pesos

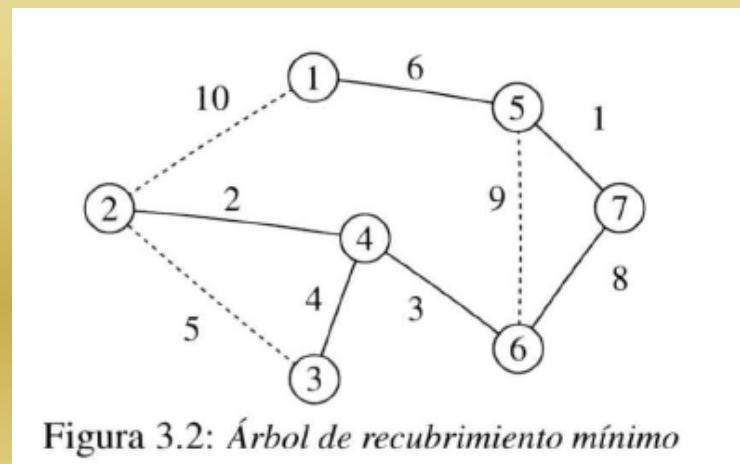


Figura 3.2: Árbol de recubrimiento mínimo

Clave: contiene la distancia mínima de los nodos del AR a cada uno de los nodos restantes					
Paso	nodoMinimo[1]	costeMinimo	min	nodo	AR
Inicio	_ 1 1 1 1 1	-1 10 ∞ ∞ 6 ∞ ∞			0
i=1			∞	2º y 5º nodo	
j=2..n	_ 1 1 1 1 5 5	-1 10 ∞ ∞ -1 ∞ ∞	10,6	2,5	AR U {1,5}
j=2..n		-1 10 ∞ ∞ -1 9 1			
i=2			∞		
j=2..n		-1 10 ∞ ∞ -1 9 -1	10,9,1	2,6,7	AR U {5,7}
j=2..n		-1 10 ∞ ∞ -1 8 -1			
i=3			∞		
j=2..n		-1 10 ∞ ∞ -1 -1 -1	10,8	2,6	AR U {7,6}
j=2..n		-1 10 ∞ 3 -1 -1 -1			
i=4			∞		
j=2..n		-1 10 ∞ -1 -1 -1 -1	10,3	2,4	AR U {6,4}
j=2..n		-1 2 4 -1 -1 -1 -1			
i=5			∞		
j=2..n		-1 -1 4 -1 -1 -1 -1	2	2	AR U {4,2}
j=2..n		-1 -1 4 -1 -1 -1 -1			
i=6			∞		
j=2..n		-1 -1 -1 -1 -1 -1 -1	4	3	AR U {4,3}
j=2..n		-1 -1 -1 -1 -1 -1 -1			

Coste:

- bucle principal $n-1$ veces y dentro de dicho bucle hay 2 bucles que se ejecutan $n-2$ veces. Implementando el grafo con una matriz adyacencia: $O(n^2)$
- Alternativa: usar listas de adyacencia junto con un montículo (el montículo almacena los candidatos pendientes): $O(a \log n)$, donde a es el nº de aristas: (disperso) $n-1 \leq a \leq n(n-1)/2$ (denso), por lo que sólo compensa si el grafo es disperso.

3.2.2 Árboles de recubrimiento mínimo: algoritmo de Kruskal

- Cambia la forma de seleccionar las aristas:
 - i. Comienza con un conjunto de aristas AR vacío, de forma que parto de n componentes conexas de 1 nodo cada una.
 - ii. En cada paso selecciona la arista más corta (independientemente de en donde esté) que no haya sido añadida a AR (selecciona las aristas por orden creciente de longitud).
 - iii. La arista se añadirá a la solución si y solo si sus nodos están en componentes conexas distintas (si estuviesen en la misma, se formaría un ciclo y la solución no sería factible).
 - iv. El algoritmo termina hasta que AR contiene $n-1$ aristas.

fun Kruskal ($G = \langle N, A \rangle$): grafo): conjunto de aristas

var

AR: conjunto de aristas

fvar

Ordenar(A) {Ordena A en pesos crecientes}



Las aristas se ordenan de menor a mayor pues así se van a seleccionar.

$n \leftarrow n^{\circ}$ nodos de N

$AR \leftarrow \emptyset$

Iniciar n conjuntos, uno con cada nodo de N

mientras AR no tenga $n-1$ aristas **hacer**

 seleccionar $\{u, v\}$ mínima

$comU \leftarrow$ buscarComponenteConexa(u)
 $comV \leftarrow$ buscarComponenteConexa(v)

 ← Calculamos en qué componente conexa se encuentra cada nodo

si $comU \neq comV$ **entonces**

 ← Función factible (comprueba si u y v están en componentes conexas distintas)

 fusionar($comU, comV$)

 ← Se fusionan ambas componentes conexas

$AR \leftarrow AR \cup \{(u, v)\}$

 ← Añado la arista a la solución

fsi

fmiéntras

dev AR

ffun

Demostración de optimalidad

La demostración de optimalidad del algoritmo de Kruskal se realiza por inducción. Se trata de demostrar de que si AR es prometedor seguirá siéndolo en cualquier fase del algoritmo cuando se le añada una arista:

Base: el conjunto vacío es prometedor.

Paso inductivo: suponemos que AR es un conjunto de aristas prometedor antes de que el algoritmo añada una nueva arista (u, v) . En ese momento, el nodo u se encuentra en una componente conexa y el nodo v en otra distinta. Sea C_i el conjunto de nodos de la componente que contiene a u . Tenemos que:

- El conjunto C_i es un subconjunto estricto del conjunto de nodos N de G , ya que al menos no incluye a v .
- AR es un conjunto prometedor tal que ninguna arista de AR sale de C_i .
- (u, v) es una de las aristas más cortas que salen de C_i , ya que las que sean estrictamente más cortas ya se han examinado, y se han añadido a AR o se han rechazado.

Entonces podemos utilizar el lema 3.2.1 ya que se cumplen sus condiciones y por lo tanto $AR \cup \{(u, v)\}$ también es prometedor.

Como AR es prometedor en todos los pasos del algoritmo, cuando el algoritmo se detenga AR contendrá una solución óptima.

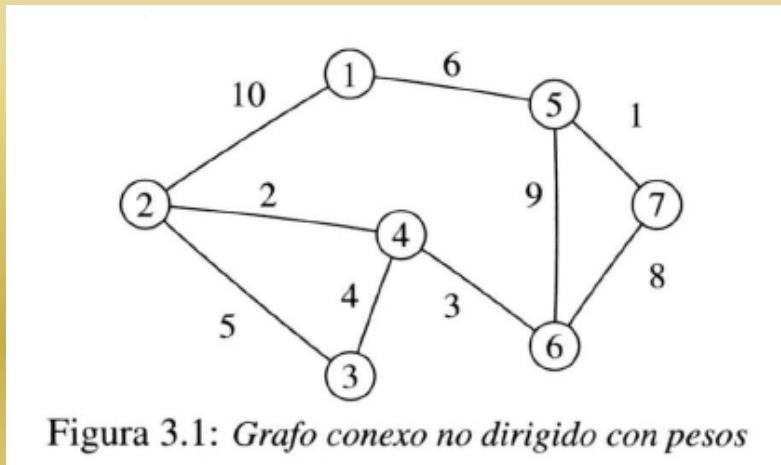


Figura 3.1: Grafo conexo no dirigido con pesos

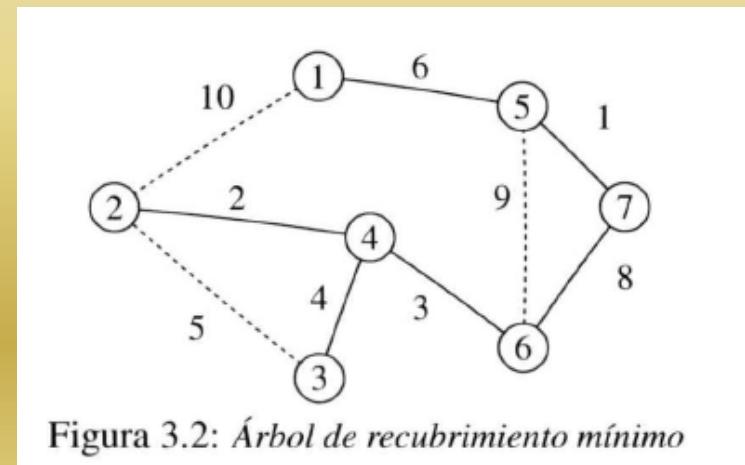


Figura 3.2: Árbol de recubrimiento mínimo

Paso	arista	Componentes conexas	AR
Inicio		{1}{2}{3}{4}{5}{6}{7}	\emptyset
1	{5,7}	{1}{2}{3}{4}{5,7}{6}	$AR \cup \{5,7\}$
2	{2,4}	{1}{2,4}{3}{5,7}{6}	$AR \cup \{2,4\}$
3	{4,6}	{1}{2,4,6}{3}{5,7}	$AR \cup \{4,6\}$
4	{3,4}	{1}{2,4,6,3}{5,7}	$AR \cup \{3,4\}$
5	{2,3}	se rechaza	
6	{1,5}	{1,5,7}{2,4,6,3}	$AR \cup \{1,5\}$
7	{6,7}	{1,5,7,2,4,6,3}	$AR \cup \{6,7\}$

Coste:

varía en una cte.

- Ordenación de aristas $O(a \log a) \uparrow O(a \log n)$, cumpliéndose (disperso) $n-1 \leq a \leq n(n-1)/2$ (denso)
- Bucle principal a veces, pero fusionar y buscarComponenteConexas son $O(1)$, por lo que el coste viene dado por la ordenación.

Grafo disperso: Kruskal $\rightarrow O(n \log n)$; Grafo denso: Prim $\rightarrow O(n^2)$

3.2.3 Camino de coste mínimo: algoritmo de Dijkstra

- Sea $G=<N,A>$ grafo dirigido, con pesos mayores o iguales que cero en sus aristas, en el que todos sus nodos son accesibles desde uno concreto considerado como origen.
- El algoritmo de Dijkstra determina la longitud del camino de coste, peso o distancia mínima que va desde el nodo origen a cada uno de los demás nodos del grafo.
 - i. Utiliza 2 conjuntos de nodos S y C :
 $S \rightarrow$ nodos ya seleccionados y cuya distancia mínima al origen ya se conoce
 $C \rightarrow$ contiene el resto de los nodos ($C=N \setminus S$).

- ii. La función de selección elegirá en cada paso el nodo de C cuya distancia al origen sea mínima.
- iii. El algoritmo termina hasta que AR contiene $n-1$ aristas.
- iv. En cada paso del algoritmo $especial[]$ es un array, de forma que $especial[i]$ es la longitud del *camino especial (es un camino que va del nodo origen a cualquier otro nodo en el que todos los nodos intermedios pertenecen a S)* más corto del nodo origen al nodo i -ésimo.
- v. A la salida, $especial[]$ contendrá las longitudes de los caminos mínimos buscadas.

- **Algoritmo (se incluye predecesor[] para almacenar el camino) :**

```

tipo VectorNat = matriz[0..n] de natural
fun Dijkstra ( $G = \langle N, A \rangle$ : grafo): VectorNat, VectorNat
    var
        especial, predecesor: VectorNat
        C: conjunto de nodos
    fvar
        C = {2, 3, ..., n}
    para i  $\leftarrow$  2 hasta n hacer
        especial[i]  $\leftarrow$  Distancia(1,i)    $\leftarrow$  Devuelve la longitud de la arista entre 2 nodos ( $\infty$  si no existe)
        predecesor[i]  $\leftarrow$  1
    fpara
    mientras C contenga más de 1 nodo hacer
        v  $\leftarrow$  nodo  $\in$  C que minimiza especial[v]    $\leftarrow$  La función seleccionar devuelve el nodo de los que quedan en C (todavía no incluido en S) con menor especial
        C  $\leftarrow$  C \{v\}    $\leftarrow$  Se elimina el nodo de C ( $\equiv$  se añade a S)
        para cada w  $\in$  C hacer
            si especial[w] > especial[v] + Distancia(v,w) entonces
                especial[w]  $\leftarrow$  especial[v] + Distancia(v,w)    $\leftarrow$  Actualizamos especial[]
                predecesor[w]  $\leftarrow$  v
            fsi
        fpara
    fmientras
    dev especial[],predecesor[]
ffun

```

Ejemplo de ejecución

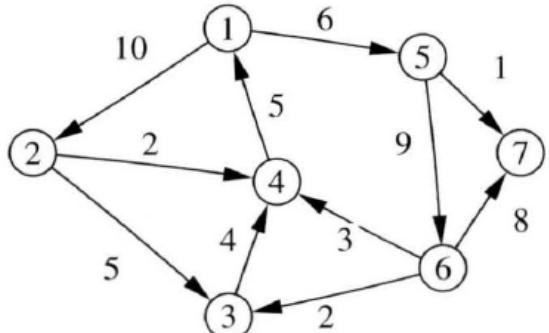


Figura 3.3: Grafo conexo dirigido con pesos

Paso	v, w	C	especial[]	predecesor[]
Inicio		{2,3,4,5,6,7}	10 ∞ ∞ 6 ∞ ∞	1 1 1 1 1 1
1	v = 5 w = 2,3,4,6,7	{2,3,4,6,7}	10 ∞ ∞ 6 15 7	1 1 1 1 5 5
2	v = 7 w = 2,3,4,6	{2,3,4,6}	10 ∞ ∞ 6 15 7	1 1 1 1 5 5
3	v = 2 w = 3,4,6	{3,4,6}	10 15 12 6 15 7	1 2 2 1 5 5
4	v = 4 w = 3,6	{3,6}	10 15 12 6 15 7	1 2 2 1 5 5
5	v = 3 w = 6	{6}	10 15 12 6 15 7	1 2 2 1 5 5

Coste (grafo n nodos y a aristas):

- **Bucle mientras** → n iteraciones. Dentro de este bucle ya 2 bucles adicionales:
 - *Bucle que calcula el nodo que minimiza especial* → $O(n)$
 - *Bucle para* → $O(n)$
- *Por lo tanto el coste es cuadrático* → $O(n^2)$
- Se puede mejorar si se evita examinar todo el array especial[] utilizando un montículo de mínimos de forma que en la raíz estaría el que minimiza especial[]. Esto permitiría obtener un coste $O((n+a)\log n)$ → *Grafo disperso: $O(n \log n)$; Grafo denso: $O(n^2 \log n)$, por lo que sólo compensa si el grafo es disperso.*

Demostración de optimalidad

La demostración de que el algoritmo de Dijkstra calcula los caminos de menor coste o longitud desde un nodo tomado como origen a los demás nodos del grafo se realiza por inducción. Se trata de demostrar que:

1. Si un nodo $i \neq 1$ está en S , entonces $\text{especial}[i]$ almacena la longitud del camino más corto desde el origen, 1, hasta el nodo i .
2. Si un nodo i no está en S , entonces $\text{especial}[i]$ almacena la longitud del camino especial más corto desde el origen, 1, hasta el nodo i .

Además, por hipótesis estos dos puntos se cumplen inmediatamente antes de añadir un nodo v al conjunto S .

Base: inicialmente sólo el nodo 1 esta en S , por lo que el punto 1 está demostrado. Para el resto de los nodos el único camino especial posible es el camino directo al nodo 1, cuya distancia es la que le asigna el algoritmo a $\text{especial}[i]$, por lo que el punto 2 también está demostrado.

Paso inductivo: como los nodos que ya están en S no se vuelven a examinar, el punto 1 se sigue cumpliendo antes de añadir un nodo v a S . Antes de poder añadir el nodo v a S hay que comprobar que $\text{especial}[v]$ almacene la longitud del camino más corto o de menor coste desde el origen hasta v . Por hipótesis, $\text{especial}[v]$ almacena la longitud del camino especial más corto, por lo que hay que verificar que dicho camino no pase por ningún nodo que no pertenece a S . Supongamos que en el camino más corto desde el origen a v hay uno o más nodos, que no son v , que no pertenecen a S . Sea x el primer nodo de este tipo. Como x no está en S , $\text{especial}[x]$ almacena la longitud del camino especial desde el origen hasta x . Como el algoritmo ha seleccionado a v antes que a x , $\text{especial}[x]$ no es menor que $\text{especial}[v]$. Por lo tanto, la distancia total hasta v a través de x es

como mínimo $\text{especial}[v]$, y el camino a través de x no puede ser más corto que el camino especial que lleva a v . Por lo tanto, cuando se añade v a S se cumple el punto 1.

Con respecto al punto 2, consideremos un nodo u que no sea v y que no pertenece a S . Cuando se añade v a S puede darse una de dos posibles situaciones: (1) $\text{especial}[u]$ no cambia porque no se encuentra un camino más corto a través de v o, (2) $\text{especial}[u]$ si cambia porque se encuentra un camino más corto a través de v y quizás de algún otro u otros nodos de S . En este caso (2), sea x el último nodo de S visitado antes de llegar a u . La longitud de ese camino será $\text{especial}[x] + \text{Distancia}(x, u)$. Se podría pensar que para calcular el nuevo valor de $\text{especial}[u]$ habría que comparar el valor anterior de $\text{especial}[u]$ con los valores de $\text{especial}[x] + \text{Distancia}(x, u)$ para todo $x \in S$, incluyendo a v . Pero esa comparación ya se hizo cuando se añadió x a S por lo que $\text{especial}[x]$ no ha cambiado desde entonces. Por lo tanto, el cálculo del nuevo valor de $\text{especial}[u]$ se puede hacer comparando solamente su valor anterior con $\text{especial}[v] + \text{Distancia}(v, u)$. Como esto es lo que hace el algoritmo, el punto 2 también se cumple cuando se añade un nuevo nodo v a S .

Al finalizar el algoritmo todos los nodos excepto uno estarán en S y el camino más corto desde el origen hasta dicho nodo es un camino especial. Por lo que queda demostrado que el algoritmo de Dijkstra calcula los caminos de menor coste o longitud desde un nodo tomado como origen a los demás nodos del grafo.

3.3 Algoritmos voraces para planificación

- Planificación de forma óptima una serie de tareas que deben llevarse a cabo por un agente o servidor (o un número determinado de ellos).
- Hay 2 variantes:
 - ❖ Minimización del tiempo del sistema
 - Planificar el orden de realización de unas tareas de forma que se minimice el tiempo total que están los clientes en el sistema (o tiempo medio de espera por cliente).
 - ❖ Planificación con plazos
 - Aquí tenemos n tareas con beneficio b_i y con una fecha tope de ejecución f_i . El objetivo es planificar la realización de algunas (o todas) tareas de forma que se minimice el beneficio obtenido (sólo se obtiene beneficio de una tarea si ésta se ejecuta antes de superar su fecha límite).

3.3.1 Minimización del tiempo del sistema

- n clientes/tareas que esperan un servicio de un determinado agente o servidor.
- t_i : **tiempo de servicio** → es el tiempo que requerirá el agente/servidor en ejecutar la tarea i -ésima (o atender al cliente i).
- **Objetivo:** seleccionar el orden de ejecución de tareas de forma que se minimice el tiempo medio de estancia/espera de los clientes en el sistema.
- **Ejemplo:** 3 clientes con tiempos de servicio $t_1=2$, $t_2=8$ y $t_3=4$. Hay 6 posibilidades de atender a los 3 clientes:

Orden de servicio	$T = \sum_{i=1}^n$ tiempo en el sistema del cliente i
1 2 3	$2 + (2 + 8) + (2 + 8 + 4) = 26$
1 3 2	$2 + (2 + 4) + (2 + 4 + 8) = 22$ ← Óptima (orden creciente de t_i)
2 1 3	$8 + (8 + 2) + (8 + 2 + 4) = 32$
2 3 1	$8 + (8 + 4) + (8 + 4 + 2) = 34$ ← Peor (orden decreciente de t_i)
3 1 2	$4 + (4 + 2) + (4 + 2 + 8) = 24$
3 2 1	$4 + (4 + 8) + (4 + 8 + 2) = 30$

- **Algoritmo voraz** → selecciona los clientes en orden creciente de tiempos de servicio (factible no es necesario).
- **Pseudocódigo** → sólo necesitaríamos un algoritmo de ordenación de los tiempos de servicio.
- **Coste** → el de la ordenación de elementos: $O(n \log n)$.

Demostración de optimalidad

Supongamos que hay n clientes numerados de 1 a n . Sea $S = s_1, s_2, \dots, s_n$ una permutación de los valores de 1 a n , y sea $e_i = t_{s_i}$ el tiempo de estancia o de servicio en el sistema del cliente i -ésimo. El tiempo total que están en el sistema todos los clientes es:

$$T(S) = e_1 + (e_1 + e_2) + (e_1 + e_2 + e_3) + \dots = ne_1 + (n-1)e_2 + (n-2)e_3 + \dots$$

$$= \sum_{k=1}^n (n-k+1)e_k$$

Supongamos que la permutación S no organiza a los clientes en orden creciente de tiempos de servicio. Entonces, habrá al menos dos enteros a y b tales que $a < b$ y $e_a > e_b$. Si se intercambia la posición de los clientes a y b se obtiene una nueva secuencia de servicio S' cuyo tiempo total en el sistema será:

$$= T(S') = (n-a+1)e_b + (n-b+1)e_a + \sum_{k=1, k \neq a,b}^n (n-k+1)e_k$$

Si restamos los tiempos de ambas permutaciones se obtiene:

$$\begin{aligned} T(S) - T(S') &= (n-a+1)(e_a - e_b) + (n-b+1)(e_b - e_a) \\ &= (n-a+1 - (n-b+1))(e_a - e_b) \\ &= (b-a)(e_a - e_b) > 0. \end{aligned}$$

Como sabemos que $a < b$ y que $e_a > e_b$ la permutación S' requiere menor tiempo que la permutación S . Es decir, cualquier permutación que no sea en orden creciente de tiempos de servicio se puede mejorar, de ahí que toda permutación cuyos clientes o tareas están ordenados por tiempos crecientes es óptima.

Queda el caso de cuando $a < b$ y $e_a = e_b$. En este caso el orden entre a y b es indiferente ya que se pueden intercambiar los valores e_a y e_b sin que se altere el valor de la suma total de tiempos.

Generalización a a agentes iguales

$$s_1 : t_1, t_{a+1}, t_{2a+1}, \dots$$

$$s_2 : t_2, t_{a+2}, t_{2a+2}, \dots$$

⋮

$$s_a : t_a, t_{2a}, t_{3a}, \dots$$

Así, al agente s_j le correspondería la secuencia de clientes $t_1^j, t_2^j, \dots, t_{n_j}^j$, siendo $t_k^j = t_{(k-1)a+j}$, para $1 \leq k \leq n_j$ y

$$n_j = \begin{cases} n \text{ div } a + 1 & \text{si } j \leq (n \text{ mod } a) \\ n \text{ div } a & \text{si } j > (n \text{ mod } a) \end{cases}$$

El tiempo total en el sistema de S será:

$$T(S) = \sum_{j=1}^a \sum_{k=1}^{n_j} (n_j - k + 1) t_{(k-1)a+j}$$

3.3.2 Planificación con plazos

- **n tareas:** cada tarea i tiene asignado un **beneficio** $b_i > 0$ y un **plazo de ejecución** $f_i > 0$.
 - Para cualquier tarea i , se obtiene el beneficio $b_i \leftrightarrow$ el trabajo se realiza no más allá de su fecha tope.
 - Las tareas se realizan en una única máquina que sólo puede realizar una tarea cada día.
- **Objetivo:** seleccionar la secuencia de tareas a realizar para que el beneficio total sea máximo.
- **Solución factible:** conjunto de tareas S para el que existe al menos una secuencia que permite que todas las tareas del conjunto se ejecuten en su plazo. Beneficio de dicha solución → suma de los beneficios de todos los trabajos que contiene: $\sum_{i \in S} b_i$
- Ejemplo: $n=4$ con los siguientes beneficios y plazos:

i	1	2	3	4
f_i	2	1	2	1
b_i	100	10	15	25

Soluciones factibles, secuencia de realización y beneficios obtenidos (la secuencia 1,4 no es factible)

Solución factible	Secuencia	Beneficio	
(1,2)	2,1	110	
(1,3)	1,3 o 3,1	115	
(1,4)	4,1	125	← Óptima
(2,3)	2,3	25	
(3,4)	4,3	40	
(1)	1	100	
(2)	2	10	
(3)	3	15	
(4)	4	25	

- **Algoritmo voraz:**

- **Función de selección:** elige los trabajos en orden decreciente de beneficios.
- **Función completable:** en cada paso, el trabajo se incorpora a la solución S , si al unir dicho trabajo a S éste sigue siendo factible.

Lema 3.3.1 *Si S es un conjunto de trabajos, entonces S es factible si y solo si la secuencia obtenida ordenando los trabajos en orden no decreciente de fechas tope es factible.*

Para demostrar este lema supongamos que S es factible, entonces existe al menos una secuencia factible de los trabajos de S . Supongamos que en esa secuencia factible el trabajo u se planifica antes que el trabajo v siendo $f_v < f_u$. Si intercambiamos los dos trabajos, el trabajo v se adelanta comenzando antes de su fecha tope. Al retrasar u a la posición de v y al ser $f_v < f_u$ y la secuencia factible, la nueva secuencia sigue siendo factible. Supongamos ahora que la secuencia $1, 2, \dots, k$ de S no es factible. Esto significa que al menos un trabajo está planificado después de su plazo. Sea l una cualquiera de esas tareas tal que $f_l \leq l - 1$. Como las tareas están ordenadas por orden no decreciente de plazos, al menos l tareas tienen como fecha final $l - 1$ o una fecha anterior. Por lo que en cualquier caso la última siempre está planificada más tarde que su fecha tope. Así, basta comprobar una secuencia en orden no decreciente de fechas para saber si un conjunto de tareas es o no factible.

*Conclusión: basta con comprobar si es factible la secuencia en orden no decreciente de fechas (de menor a mayor f_i) para saber si un conjunto de tareas es o no factible.
→ No hace falta comprobar todas permutaciones/secuencias.*

Descripción de alto nivel

(los índices i de $F[i]$ contienen las tareas ordenadas de mayor a menor beneficio)

```

fun PlanificacionPlazos (F[1..n], n): Vector[1..n] de natural
    S ← {1}
    para i = 2 hasta n hacer
        si los trabajos en S ∪ {i} constituyen una secuencia factible entonces
            S ← S ∪ {i}
        fsi
    fpara
    dev S
ffun

```

Demostración de optimalidad

Se trata de demostrar que el algoritmo voraz que considera los trabajos en orden decreciente de beneficios, siempre que el conjunto de trabajos sea una solución factible, encuentra una planificación óptima.

Supongamos que se quieren ejecutar un conjunto I y un conjunto J de trabajos o tareas. El conjunto I consta de las tareas $\{g, h, i, k, l\}$ y el conjunto J consta de las tareas $\{l, m, o, g, p, q, i\}$. Supongamos también que el conjunto J es óptimo. Sean S_I y S_J secuencias factibles de los respectivos conjuntos, que podrían incluir huecos en su planificación:

$S_I :$	g	h	i	k	l		
$S_J :$	l	m	o	g	p	q	i

Se pueden reorganizar los trabajos de S_I y S_J de manera que se obtienen las secuencias S''_I y S''_J que también son factibles, en las que los trabajos comunes a ambas secuencias se planifican en la misma unidad de tiempo. También en estas secuencias podría haber huecos:

$S''_I :$	k	h		g	l		i
$S''_J :$	p	m	o	g	l	q	i

Supongamos que un trabajo a aparece en las 2 secuencias factibles S_I y S_J , planificado respectivamente en los tiempos t_I y t_J . En el caso de que $t_I = t_J$ no hay cambios posibles. Supongamos que $t_I < t_J$. Como la secuencia S_J es factible, el plazo para la tarea a no es anterior a t_I y se atrasa la tarea a del tiempo t_I al tiempo t_J en su secuencia S_I . Se pueden dar dos situaciones:

- Si hay un hueco en la secuencia S_I en la unidad de tiempo t_J se pasa la tarea a a dicho hueco.
- Si ya hay una tarea b planificada en ese tiempo t_J en S_I , se modifica S_I intercambiando las tareas a y b en la secuencia.

La secuencia resultante sigue siendo factible ya que a se sigue ejecutando antes de su plazo y en el caso de que exista b se adelanta. Ahora a está planificada en ambas secuencias en la unidad de tiempo t_J . En el caso de que $t_I > t_J$ se realiza lo mismo pero en este caso con la secuencia S_J . Una vez realizado este procedimiento con la tarea a , ésta se queda ya en dicha unidad de tiempo.

Si las secuencias S_I y S_J tienen r tareas en común, tras un máximo de r modificaciones de S_I o S_J , las tareas comunes a ambas secuencias estarán planificadas al mismo tiempo en S''_I y S''_J . Si $I \neq J$ las secuencias resultantes S''_I y S''_J pueden no ser iguales. Supongamos que en una determinada unidad de tiempo la tarea planificada en S''_I es distinta de la planificada en S''_J :

- Si hay una tarea a en S''_I y en la misma unidad de tiempo en S''_J hay un hueco, entonces a no pertenece a J . El conjunto $J \cup \{a\}$ sería factible, ya que se podría poner a en el hueco y el beneficio sería mayor que en J . Pero por hipótesis partimos de que J es óptimo, por lo que no es posible.
- Si hay una tarea b en S''_J y en la misma unidad de tiempo en S''_I hay un hueco, el conjunto $J \cup \{b\}$ sería factible, por lo que el algoritmo habría incluido b en I . Como el algoritmo no lo hizo entonces tampoco es posible.
- El tercer caso es que una tarea a esté planificada en S''_I y en la misma unidad de tiempo en S''_J hay una tarea distinta b . En este caso a no pertenece a J y b no pertenece a I . Las posibilidades son:
 - Si $b_a > b_b$ se podría sustituir a por b en J y mejorar el beneficio. Pero esto no es posible ya que J es óptima.
 - Si $b_a < b_b$ el algoritmo voraz habría seleccionado a antes de considerar a b puesto que $(I \setminus \{a\}) \cup \{b\}$ sería factible. Esto no es posible porque el algoritmo voraz no incluyó a b en I .
 - Si $b_a = b_b$ aportan la misma ganancia en las secuencias.

Por lo que para toda unidad de tiempo de las secuencias S''_I y S''_J o no planifican tareas, o planifican la misma, o bien planifican dos tareas distintas con igual beneficio. El beneficio total de I es por lo tanto igual al beneficio del conjunto óptimo J , así que I también es óptimo.

Algoritmo detallado:

tipo VectorNat = matriz[0..n] de natural

fun PlanificacionPlazosDetallado (f: VectorNat, n: natural): VectorNat, natural

var
 S: VectorNat (El array $f[i]$ almacena las fechas tope en orden decreciente de beneficios)

fvar

$S[0] \leftarrow 0$ {elemento centinela para facilitar la inserción}

$S[1] \leftarrow 1$ {se incluye el trabajo 1 que es el de máximo beneficio}

$k \leftarrow 1$ {k: nº de elementos en S}

para i = 2 **hasta** n **hacer** ← Como mucho podré planificar n tareas

$r \leftarrow k$ {se busca una posición válida para i}

mientras ($f[S[r]] > f[i]$) \wedge ($f[S[r]] \neq r$) **hacer**

$r \leftarrow r - 1$

fmientras

si ($f[S[r]] \leq f[i]$) \wedge ($f[i] > r$) **entonces** ← Para cada tarea i-ésima el algoritmo comprueba que se pueda insertar sin desplazar a las tareas ya planificadas mas allá de su plazo. Si es posible inserta dicha tarea y sino la descarta.

para q = k **hasta** r + 1 **incr** = -1 **hacer**

$S[q+1] \leftarrow S[q]$ ← Corro las tareas hacia la derecha para hacer hueco

fpara

$S[r+1] \leftarrow i$ ← Añado la tarea en la posición r+1

$k \leftarrow k + 1$

fsi

fpara

dev S, k

ffun

Ejemplo de ejecución

- $n=5$
- $b_i = (20, 15, 10, 5, 1)$
- $f_i = (2, 2, 1, 3, 3)$

Paso	k	i	S
Inicialización	1		1 _ _ _ _
1		2	
	2	r = 1	1 2 _ _ _
2		3	
		r = 2	1 2 _ _ _
3		4	
	3	r = 2	1 2 4 _ _
4		5	
		r = 3	

← El elemento $i=3$ lo descarta porque la solución no sería factible

← Planificación óptima: 1,2,4
Beneficio: 40

Coste:

- Ordenación de $f[n]$ en orden decreciente de beneficios $\rightarrow O(n \log n)$
- **Caso peor:** todas las tareas en f también están ordenadas por orden decreciente de plazos y se pueden incluir en la planificación. En este caso para cada tarea i , el algoritmo evalúa las $i-1$ tareas ya seleccionadas, encuentra su ubicación y realiza los desplazamientos oportunas (bucle anidado) $\rightarrow O(n^2)$

• Algoritmo mejorado:

- Se consigue mejorar la eficiencia si **cada tarea** (seleccionada en orden decreciente de beneficios) **se programa lo más tarde posible** (de acuerdo a su plazo de ejecución y siempre que haya hueco).
- Empezando con una planificación vacía, para cada tarea $i \in S$ se planifica i en el instante t , siendo t el mayor entero tal que $1 \leq t \leq \min\{n, f_i\}$ y la tarea que se ejecutará en t no se ha decidido todavía.

Demostración de optimalidad

Veamos la demostración de que este método de determinar la factibilidad de una solución parcial conduce a una solución óptima. El que cada tarea i se añada a la planificación en el instante de tiempo más tardío posible, se puede formalizar indicando que el día elegido será:

$$t(i) = \max\{t \mid 1 \leq t \leq \min\{n, f_i\} \wedge (\forall j : 1 \leq j < i : t \neq t(j))\}.$$

Cuando se intenta añadir una nueva tarea, la secuencia que se está construyendo contiene al menos un hueco. Supongamos que no se puede añadir una tarea i cuyo plazo sea f_i . Esto sólo puede ocurrir si todas las posiciones desde la 1 hasta la $\min\{n, f_i\}$ están ya ocupadas. Sea $s > \min\{n, f_i\}$ el mayor entero tal que la posición $t = s$ está vacía. La planificación incluye por lo tanto $s - 1$ tareas, ninguna tarea con plazo s y quizás otras con plazos posteriores a s . La tarea que se quiere añadir también tiene un plazo anterior a s . Como la planificación contiene al menos s tareas cuyos plazos son $s - 1$ o anteriores, la última tarea llegará tarde, lo que demuestra el “solo si” de la hipótesis.

Para implementar esta prueba de factibilidad definimos $\text{libre}(i) = \max\{t \leq i \mid t \text{ libre}\}$, es decir, es el primer predecesor libre de i . De esta manera se definen conjuntos de posiciones, de forma que las posiciones i y j están en el mismo conjunto si y solo si $\text{libre}(i) = \text{libre}(j)$. Se define además una posición ficticia 0 que siempre estará libre. Así, la tarea i debería realizarse en el día $\text{libre}(f_i)$ ya que representa el último día libre que respeta su plazo.

Los pasos principales del algoritmo mejorado son:

- Inicialmente cada posición o instante de tiempo $0, 1, 2, 3, \dots, p$ está en un conjunto diferente y $\text{libre}(\{i\}) = i$, $0 \leq i \leq p$
- Si se quiere añadir una tarea con plazo f se busca el conjunto que contenga a f . Sea K dicho conjunto. Si $\text{libre}(K) = 0$ se rechaza la tarea; en caso contrario se realizan las siguientes acciones:
 - Se asigna la tarea al instante de tiempo $\text{libre}(K)$.
 - Se busca el conjunto que contenga $\text{libre}(K) - 1$. Sea L dicho conjunto.
 - Se fusionan K y L . El valor de $\text{libre}()$ para este nuevo conjunto es el valor que tenía $\text{libre}(L)$.

Pseudocódigo algoritmo mejorado:

```

tipo VectorNat = matriz[0..n] de natural
fun PlanificacionPlazosMejorado (f: VectorNat, n: natural): VectorNat, natural
    var
        S, libre: VectorNat
    fvar
        p  $\leftarrow$  min(n, max{f[i] | 1  $\leq$  i  $\leq$  n })
        para i = 0 hasta n hacer
            S[i]  $\leftarrow$  0
            libre[i]  $\leftarrow$  i
            Iniciar conjunto i  $\leftarrow$  Inicializo n conjuntos disjuntos (partición)
        fpara
        para i = 1 hasta n hacer
            k  $\leftarrow$  buscar(min(p,f[i]))
            pos  $\leftarrow$  libre(k)
            si pos  $\neq$  0 entonces  $\leftarrow$  Esto comprueba si hay hueco para colocar la tarea i
                S[pos]  $\leftarrow$  i
                l  $\leftarrow$  buscar(pos - 1)  $\leftarrow$  busco el conjunto anterior
                libre[k]  $\leftarrow$  libre[l]
                fusionar(k,l) {asignar la etiqueta k o l}
            fsi
        fpara
        k  $\leftarrow$  0
        para i = 1 hasta n hacer
            si S[i] > 0 entonces
                k  $\leftarrow$  k + 1
                S[k]  $\leftarrow$  S[i]
            fsi
        fpara
        dev S, k
    ffun

```

Comprimo S[] eliminando los ceros

\leftarrow Fusiono los conjuntos k y l

- Inicialmente cada posición o instante de tiempo $0, 1, 2, 3, \dots, p$ está en un conjunto diferente y $libre(\{i\}) = i$, $0 \leq i \leq p$
- Si se quiere añadir una tarea con plazo f se busca el conjunto que contenga a f . Sea K dicho conjunto. Si $libre(K) = 0$ se rechaza la tarea; en caso contrario se realizan las siguientes acciones:
 - Se asigna la tarea al instante de tiempo $libre(K)$.
 - Se busca el conjunto que contenga $libre(K) - 1$. Sea L dicho conjunto.
 - Se fusionan K y L . El valor de $libre()$ para este nuevo conjunto es el valor que tenía $libre(L)$.

Ejemplo de ejecución

- $n=5$, $b_i=(20,15,10,5,1)$ y $f_i=(2,2,1,3,3)$

Paso	p	k	pos	l	libre[]	S
Inicialización	3				{0} {1} {2} {3} {4} {5}	0 0 0 0 0
1		2	2	1	{0} {1, 2} {3} {4} {5}	0 1 0 0 0
2		2	1	0	{0, 1, 2} {3} {4} {5}	2 1 0 0 0
3		1	0		{0, 1, 2} {3} {4} {5}	2 1 0 0 0
4		3	3	2	{0, 1, 2, 3} {4} {5}	2 1 4 0 0
5		3	0		{0, 1, 2, 3} {4} {5}	2 1 4 0 0
Comprimir S[]					{0, 1, 2, 3} {4} {5}	2 1 4 0 0

Coste:

- Ordenación de $f[n]$ en orden decreciente de beneficios $\rightarrow O(n \log n)$
- Fase de inicialización $\rightarrow O(n)$
- $2n$ operaciones buscar $O(1)$ y n operaciones fusionar $O(1) \rightarrow O(n)$
- Total $\rightarrow O(n \log n)$

3.4 Almacenamiento óptimo en un soporte secuencial (similar a minimización del tiempo del sistema)

- Tenemos n programas p_1, p_2, \dots, p_n de longitudes $\ell_1, \ell_2, \dots, \ell_n$.
- Contamos con un soporte secuencial de longitud L , lo que indica que se podrán almacenar los programas en el soporte si y sólo si la suma de sus longitudes no supera L .
- Suposición: para poder acceder a un programa p_i tendremos que posicionar el mecanismo de acceso al comienzo del soporte.
- Si los programas están almacenados en el orden $I = i_1, i_2, \dots, i_n$, el tiempo necesario para recuperar el programa i_j es $t_j = \sum_{1 \leq k \leq j} \ell_{ik}$
de donde el tiempo total para acceder a los n programas para el orden I :

$$D(I) = \sum_{1 \leq j \leq n} t_j = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} \ell_{ik}$$

- Objetivo: encontrar una secuencia de almacenamiento de los programas (I) que minimice el tiempo medio de acceso $D(I)/n$ (equivale a minimizar el tiempo total)
- Esto se consigue si seleccionar elige los programas en orden creciente de longitudes (porque al posicionarse siempre desde el principio, los primeros son los que más van a influir en el tiempo de acceso del resto).

Funciones del esquema general

- **Resolución de un problema de forma óptima:** almacenar una serie de programas en un soporte secuencial de manera que el tiempo medio de acceso sea mínimo.
- **Conjunto inicial de candidatos:** los n programas.
- **Conjunto de candidatos seleccionados:** inicialmente el conjunto de candidatos ya escogidos está vacío.
- **Solución:** si la suma de las longitudes de todos los programas es como mucho L se alcanzará la solución cuando los n programas se hayan seleccionado en el orden adecuado y se hayan almacenado.
- **Factible:** si la suma de las longitudes de todos los programas es como mucho L el conjunto resultante siempre es factible.
- **La función de selección** se traduce en la selección de los programas en orden no decreciente de longitudes.
- **La función objetivo** no aparece explícitamente en el algoritmo

Algoritmo voraz

```

fun AlmacenaProgramasSoporteSec (c: conjuntoCandidatos): conjuntoCandidatos
    Ordenar c en orden no decreciente de longitudes
    sol ← ∅
    mientras c ≠ ∅ hacer
        x ← seleccionar(c)      ← En orden creciente de longitudes
        c ← c - {x}
        sol ← sol ∪ {x}
    fmientras
    devolver sol
ffun

```

Coste:

- La implementación del conjunto de los programas y del conjunto solución se puede hacer en un array.
- Coste de ordenar los programas de menor a mayor coste → $O(n \log n)$
- Coste del bucle mientras → $O(n)$
- Total → $O(n \log n)$

Demostración de optimidad y generalización a n soportes secuenciales: similar a minimización tiempo del sistema.

3.5 Problema de la mochila con objetos fraccionables

- **Datos del problema:**
 - n : número de objetos disponibles.
 - M : peso máximo admitido por la mochila.
 - $p = (p_1, p_2, \dots, p_n)$ pesos de los objetos.
 - $v = (v_1, v_2, \dots, v_n)$ valores/beneficios de los objetos.

➤ **Objetivo:** llenar la mochila (con objetos o fracciones de objetos) de manera que se maximice el valor total de los objetos almacenados en ésta.

- **Formulación matemática:**

$$\text{Maximizar } \sum_{i=1..n} x_i v_i, \text{ sujeto a la restricción } \sum_{i=1..n} x_i p_i \leq M \text{ y } 0 \leq x_i \leq 1$$

- **Esquema voraz:**
 - **Conjunto de candidatos** → objetos o fracciones de objetos.
 - **Función objetivo** → valor total de los objetos almacenados en la mochila.
 - **Conjunto solución** → array n de reales: $x = (x_1, x_2, \dots, x_n)$, $0 \leq x_i \leq 1$. Al ser los objetos fraccionables, la solución óptima ha de llenar exactamente la mochila.
 - **¿Función de selección?**

Función de selección → 3 posibilidades:

- 1) Seleccionar el objeto más valioso de los restantes
- 2) Seleccionar el objeto de menos peso de los restantes
- 3) Seleccionar el objeto con mayor valor por unidad de peso de los restantes.

Ejemplo: $n=3$, $M=20$, $(p_1, p_2, p_3) = (18, 15, 20)$ y $(v_1, v_2, v_3) = (25, 24, 15)$

Función selección	x_i	$\sum_{i=1}^n x_i v_i$	$\sum_{i=1}^n x_i p_i$
maximizar v_i	1 2/15 0	$25 + 3.2 = 28.2$	$18 + 2 = 20$
minimizar p_i	0 10/15 1	$16 + 15 = 31$	$10 + 10 = 20$
maximizar v_i/p_i	0 1 5/10	$24 + 7.5 = 31.5$	$15 + 5 = 20$

Valor por unidad de peso de los objetos: $(p_1/v_2, p_1/v_2, p_3/v_3) = (1.388, 1.6, 1.5)$

Algoritmo voraz

```

tipo VectorNat = matriz[0..n] de natural
tipo VectorRea = matriz[0..n] de real
fun MochilaObjetosFraccionables (p: VectorNat, v: VectorNat, M: natural): VectorRea
    var
        x: VectorRea
        peso: natural
    fvar
        Ordenar objetos en orden no creciente de  $v_i/p_i$ 
        peso  $\leftarrow$  0
        para i = 1 hasta n hacer
            x[i]  $\leftarrow$  0    $\leftarrow$  La solución inicial no selecciona ningún objeto o fracción
        fpara
        mientras peso < M hacer
            i  $\leftarrow$  mejor objeto de los restantes    $\leftarrow$  Función seleccionar
            si peso + p[i]  $\leq$  M entonces    $\leftarrow$  Factible: comprueba si el siguiente objeto cabe entero
                x[i]  $\leftarrow$  1    $\leftarrow$  Si cabe, lo meto entero
                peso  $\leftarrow$  peso + p[i]
            sino
                x[i]  $\leftarrow$  (M - peso)/ p[i]
                peso  $\leftarrow$  M
            fsi
        fmientras
        dev x
    ffun
    
```

Si no cabe, meto la fracción correspondiente.

Coste:

- Coste de ordenar los objetos en orden decreciente de $v_i/p_i \rightarrow O(n \log n)$
- Coste del resto $\rightarrow O(n)$
- Total $\rightarrow O(n \log n)$

• Demostración de optimalidad

Se trata de comprobar que efectivamente la función de selección propuesta conduce a que el algoritmo calcule una solución óptima.

Teorema 3.5.1 Si los objetos se seleccionan en orden decreciente de v_i/p_i es decir $v_1/p_1 \geq v_2/p_2 \geq \dots \geq v_n/p_n$, entonces el algoritmo voraz anterior encuentra una solución óptima.

Supongamos que los objetos están ordenados en orden decreciente de v_i/p_i . Sea $X = (x_1, x_2, \dots, x_n)$ la solución calculada por el algoritmo voraz. Si todos los x_i son iguales a 1, entonces la solución es óptima. En caso contrario, sea j el menor índice tal que $x_j < 1$. Del funcionamiento del algoritmo se sigue que $x_i = 1$ cuando $1 \leq i < j$, que $x_i = 0$ cuando $j < i \leq n$, que $0 \leq x_j < 1$, y que $\sum_{i=1}^n x_i p_i = M$. Sea $V(X) = \sum_{i=1}^n x_i v_i$ el valor de la solución X.

Sea $Y = (y_1, y_2, \dots, y_n)$ una solución factible cualquiera. Se trata de comparar X con Y para demostrar que $V(X) \geq V(Y)$ de lo que se deduce que X es óptima.

Si Y es factible, entonces $\sum_{i=1}^n y_i p_i \leq M$ y por lo tanto $\sum_{i=1}^n (x_i - y_i)v_i \geq 0$. Sea $V(Y) = \sum_{i=1}^n y_i v_i$ el valor de la solución Y,

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

tenemos tres posibilidades a considerar:

- Cuando $i < j$ entonces $x_i = 1$ y por lo tanto $x_i - y_i \geq 0$, mientras que $v_i/p_i \geq v_j/p_j$ debido a la ordenación de los objetos.
- Cuando $i > j$ entonces $x_i = 0$ y por lo tanto $x_i - y_i \leq 0$, mientras que $v_i/p_i \leq v_j/p_j$.
- Si $i = j$ entonces $v_i/p_i = v_j/p_j$.

Por lo que en todos los casos se tiene que $(x_i - y_i)(v_i/p_i) \geq (x_i - y_i)(v_j/p_j)$ por lo que:

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

Como $\frac{v_j}{p_j} > 0$ porque todos los valores y pesos son positivos y además se cumple que:

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

se tiene:

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

Queda demostrado que ninguna solución factible puede tener un valor mayor que $V(X)$, por lo que X es una solución óptima.

3.6 Mantenimiento de la conectividad

- Operadora de comunicaciones de 8 nodos conectados entre sí mediante fibra óptica .
 - Cada conexión tiene un coste $c(i,j) = (i \times j) \bmod 6$.
- **Objetivo:** reducir las conexiones manteniendo la conectividad de la red a coste mínimo.

Problema del cálculo del ARM → Algoritmo de Prim o Kruskal

3.8 Problema del robot desplazándose en un circuito

- Robot R que dispone de una batería de N unidades de energía.
- Se tiene que desplazar desde un punto origen hasta un punto de salida del circuito.
- En el circuito se puede encontrar obstáculos O que serán infranqueables.
- El paso de una casilla infranqueable supone un gasto total de energía igual al indicado por ésta.

➤ **Objetivo:** encontrar el camino hasta la salida con un consumo de energía mínimo.

Problema del cálculo del camino de coste mínimo → Algoritmo de Dijkstra

O	S	O	2	1
3	1	3	1	1
1	6	6	O	6
1	2	O	R	4
7	1	1	2	6

3.7 Problema de mensajería urgente

- Un transportista tiene que ir desde un origen a un destino y dispone de un vehículo que puede hacer ***n kilómetros sin repostar***.
- Se conocen las distancias entre las **G gasolineras** en todas las posibles rutas **DG[1..G-1]**, siendo $DG[i]$ el número de kilómetros que hay entre la gasolinera $i-1$ e i .
- **Objetivo:** Encontrar la ruta que tiene que seguir (en qué gasolineras debe parar) para repostar el menor número de veces posible.

Elementos del esquema voraz:

- ✓ **Conjunto inicial de candidatos** → las G gasolineras
- ✓ **Conjunto de candidatos seleccionados** → inicialmente está vacío
- ✓ **Solución** → se alcanzará la solución cuando se llegue al destino repostando según indique la función de selección.
- ✓ **Factible** → si la distancia entre las gasolineras de la solución es menor o igual que n , el conjunto resultante es siempre factible.
- ✓ **Función de selección** → se traduce en recorrer el mayor nº de kilómetros sin repostar.
→ **Tratará de recorrer el mayor nº de kilómetros in repostar, tratando de ir desde cada gasolinera en la que se pare a repostar a la más lejana posible, hasta llegar al destino.**

Demostración de optimalidad

Se trata de demostrar que esta estrategia voraz conduce a una solución óptima.

Sea X la solución de la estrategia voraz anterior y sean x_1, x_2, \dots, x_s , las gasolineras en las que el algoritmo decide parar a repostar en dicha solución. Sea Y otra solución compuesta por otro conjunto de gasolineras y_1, y_2, \dots, y_t . Sea N el número total de kilómetros entre el punto de recogida y el punto de entrega, y sea $K[i]$ la distancia en kilómetros recorrida por el vehículo hasta la gasolinera i , siendo $1 \leq i \leq G - 1$. Se tiene:

$$K[i] = \sum_{k=1}^i DG[k] \text{ y } K[G-1] = N$$

Se trata de demostrar que $s \leq t$, siendo s el número de gasolineras devueltas por el algoritmo voraz y t el número de gasolineras de cualquier otra solución, ya que lo que se quiere minimizar es el número de paradas. Para ello, bastará con demostrar que $x_k \geq y_k$ para todo k . Como X e Y son dos soluciones distintas, sea k el primer índice tal que $x_k \neq y_k$. Sin pérdida de generalidad podemos suponer que $k = 1$ ya que hasta x_{k-1} los viajes son iguales y en la gasolinera x_{k-1} en ambas soluciones se llenan los depósitos.

Siguiendo la estrategia voraz propuesta, si $x_1 \neq y_1$ entonces $x_1 > y_1$ ya que x_1 es la gasolinera más alejada a la que se puede viajar sin repostar. Podríamos decir también que $x_2 \geq y_2$ ya que x_2 es la gasolinera más alejada a la que se puede viajar desde x_1 sin repostar. Para probarlo, supongamos por reducción al absurdo que y_2 fuera estrictamente superior que x_2 . Para que en la solución Y se consiga ir desde y_1 a y_2 es que hay menos de n km. entre ellas: $K[y_2] - K[y_1] < n$, por lo que entonces desde x_1 hasta y_2 también hay menos de n km ya que $K[y_1] < K[x_1]$. En este caso, el algoritmo no hubiera escogido x_2 como siguiente gasolinera a x_1 sino y_2 , ya que busca la gasolinera más alejada entre las que puede llegar.

Se repite el proceso y se va obteniendo que $x_k \geq y_k$ para todo k , hasta llegar a la ciudad destino, lo que demuestra que la estrategia voraz consigue una solución óptima.

Algoritmo detallado:

tipo Vector = matriz[0..G-1] de booleano

fun EntregaExpresMinimasParadas (DG: Vector[1..G-1] de natural, n,G: natural): Vector

var

solucion: Vector

El vector de booleanos almacenará la solución: vector[i]=F no paro en la gasolinera i-ésima, vector[i]=V, sí paro en dicha gasolinera)

fvar

para i = 1 **hasta** G-1 **hacer**

solucion[i] \leftarrow falso

Vector solución inicial, todos los elementos a falso
(ninguna gasolinera escogida)

fpara

i \leftarrow 0

contKm \leftarrow 0

repetir

repetir

i \leftarrow i + 1

contKm \leftarrow contKm + DG[i]

Va sumando los Km de cada gasolinera

hasta (contKm > n) \vee (i = G-1)

si contKm > n **entonces**

i \leftarrow i - 1

solucion[i] \leftarrow cierto

contKm \leftarrow 0

Bucle voraz: se detendrá cuando el cuentaKm supera n (autonomía del vehículo) o cuando ya no hay mas gasolineras

fsi

hasta i = G-1

Me detengo en la gasolinera anterior
(i-1) y reseteo el cuentaKm

dev solucion[]

Coste: se ejecutan 2 bucles “anidados”, sin embargo usan el mismo contador i desde 1 hasta G-1, por lo que con todas las instrucciones elementales $\rightarrow O(G)$.

ffun

3.9 Asistencia a incidencias

- Empresa que debe atender las incidencias de sus clientes lo más rápido posible.
- Cada jornada efectúa n salidas y sabe de antemano el tiempo que va a llevar atender cada una de las incidencias.
- **Objetivo:** minimizar el tiempo medio de espera de sus clientes
→ *Similar al problema de Minimización del tiempo en el sistema (3.3.1).*