Ramificación y poda

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Enero 2013

Bibliografía

- R. Neapolitan y K. Naimipour. *Foundations of Algorithms using C++ pseudocode*. Tercera edición. Jones and Bartlett Publishers, 2004.
  Capítulo 6
- E. Horowitz, S. Sahni y S. Rajasekaran. *Computer Algorithms*. Tercera edición. Computer Science Press, 1998.
  Capítulo 8
- N. Martí Oliet, Y. Ortega Mallén y A. Verdejo. *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. Segunda edición, Garceta, 2013.
  Capítulo 15

Búsqueda en el espacio de soluciones

- No siempre podemos utilizar técnicas como divide y vencerás, método voraz o programación dinámica para lograr soluciones *eficientes*.
- El último recurso que nos queda para resolver nuestro problema es aplicar la "fuerza bruta".
- Realizar una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una que satisfaga los criterios exigidos.

Búsqueda en el espacio de soluciones

- No siempre podemos utilizar técnicas como divide y vencerás, método voraz o programación dinámica para lograr soluciones *eficientes*.
- El último recurso que nos queda para resolver nuestro problema es aplicar la "fuerza bruta".
- Realizar una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una que satisfaga los criterios exigidos.
- Esta búsqueda exhaustiva resultará impracticable si el cardinal del conjunto de soluciones posibles es muy grande, lo cual ocurre muy a menudo.
- Es imprescindible intentar estructurar el espacio a explorar, tratando de descartar en bloque posibles soluciones no satisfactorias.

Búsqueda en el espacio de soluciones

- Consideraremos problemas cuyas soluciones sean construibles por etapas.
- Una solución será expresable en forma de n-tupla $(x_1, \ldots, x_n)$, donde cada $x_i \in S_i$ representa la decisión tomada en la etapa i-ésima.

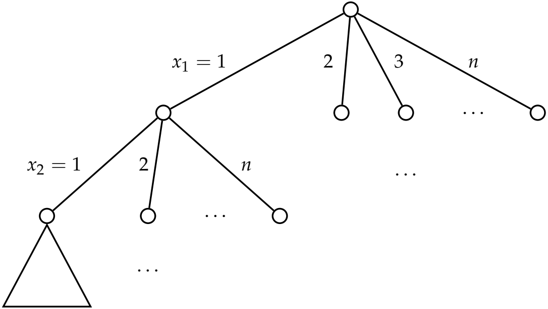
Búsqueda en el espacio de soluciones

- Consideraremos problemas cuyas soluciones sean construibles por etapas.
- Una solución será expresable en forma de n-tupla $(x_1, \ldots, x_n)$, donde cada $x_i \in S_i$ representa la decisión tomada en la etapa i-ésima.
- Además, una solución habrá de minimizar, maximizar o simplemente satisfacer una cierta función criterio.
- Se establecen dos categorías de restricciones:
  - las restricciones explícitas, que indican los conjuntos S_i a los que pertenecen cada una de las componentes de una tupla solución;
  - las restricciones implícitas, que son las relaciones que se han de establecer entre las componentes de la tupla solución para satisfacer la función criterio.

Búsqueda en el espacio de soluciones

- Consideraremos problemas cuyas soluciones sean construibles por etapas.
- Una solución será expresable en forma de n-tupla $(x_1, \ldots, x_n)$, donde cada $x_i \in S_i$ representa la decisión tomada en la etapa i-ésima.
- Además, una solución habrá de minimizar, maximizar o simplemente satisfacer una cierta función criterio.
- Se establecen dos categorías de restricciones:
  - las restricciones explícitas, que indican los conjuntos S_i a los que pertenecen cada una de las componentes de una tupla solución;
  - las restricciones implícitas, que son las relaciones que se han de establecer entre las componentes de la tupla solución para satisfacer la función criterio.
- El espacio de soluciones estará formado por el conjunto de tuplas que satisfacen las restricciones explícitas, y se puede estructurar como un árbol de exploración.

Búsqueda en el espacio de soluciones



Búsqueda en el espacio de soluciones

- Se utilizan funciones de poda o tests de factibilidad que permiten determinar cuándo una tupla parcial nunca va a conducir a una solución satisfactoria, por lo que es inútil seguir buscando a partir de ella.
- Una vez definido el árbol de exploración, el algoritmo para resolver el problema consistirá en realizar un recorrido del árbol en cierto orden, hasta encontrar la primera solución, o bien recorrer el árbol completo (salvo las zonas podadas) para obtener todas las soluciones o la solución óptima deseada.
- Durante el proceso, para cada nodo se irán generando sus sucesores; así denominaremos nodos vivos a aquellos para los cuales todavía no se han generado todos sus hijos; nodos en expansión a aquellos cuyos hijos están siendo generados; y nodos muertos a aquellos que no pueden ser expandidos, bien porque no hayan superado el test de factibilidad, o bien porque todos sus hijos ya han sido generados.

Búsqueda en el espacio de soluciones

- Existen dos tipos de recorridos:

  Vuelta atrás: el recorrido se realiza en profundidad, de forma que los nodos vivos se gestionan mediante una pila. El método resulta sencillo y eficiente en espacio.

  Ramificación y poda: corresponde a una búsqueda más "inteligente", donde en cada momento se expande el nodo vivo más "prometedor", de forma que los nodos vivos se gestionan mediante una cola con prioridad.

Búsqueda en el espacio de soluciones

- Existen dos tipos de recorridos:

  Vuelta atrás: el recorrido se realiza en profundidad, de forma que los nodos vivos se gestionan mediante una pila. El método resulta sencillo y eficiente en espacio.

  Ramificación y poda: corresponde a una búsqueda más "inteligente", donde en cada momento se expande el nodo vivo más "prometedor", de forma que los nodos vivos se gestionan mediante una cola con prioridad.

- El coste en el caso peor de todos estos algoritmos de búsqueda exhaustiva está en el orden del tamaño del espacio de soluciones, que suele ser cuando menos exponencial. Su utilidad práctica reside en la efectividad de las funciones de poda que se apliquen: cuanto más elaboradas sean, más nodos no factibles se detectarán y mayor proporción de árbol se podará. Sin embargo, si se vuelven demasiado sofisticadas, su coste de aplicación (a cada nodo del árbol) puede no llegar a compensar la poda obtenida.

Esquema general de vuelta atrás

Cuando se realiza una búsqueda en profundidad y se llega a un nodo muerto, hay que deshacer la última decisión tomada, para optar por la siguiente alternativa (como cuando en un laberinto se llega a un callejón sin salida).

```
proc vuelta-atrás(sol : tupla, e k : nat)
    preparar-recorrido-nivel(k)
    mientras ¬último-hijo-nivel(k) hacer
        sol[k] := siguiente-hijo-nivel(k)
        si es-solución?(sol, k) entonces
            tratar-solución(sol)
        si no
            si es-completable?(sol, k) entonces
                vuelta-atrás(sol, k + 1)
            fsi
        fsi
    fmientras
fproc
```

Vuelta atrás con marcaje

Se puede optimizar el test de factibilidad por el método de asociar a cada nodo cierta información que corresponde a "cálculos parciales" de dichos tests.

Se utilizan parámetros adicionales (marcadores) de entrada/salida que equivalen a variables globales y por tanto suponen un pequeño incremento del coste en espacio.

```
proc vuelta-atrás-marcaje(sol : tupla, e k : nat, m : marcador)
    preparar-recorrido-nivel(k)
    mientras ¬último-hijo-nivel(k) hacer
        sol[k] := siguiente-hijo-nivel(k)
        m := marcar(m, sol[k])
        si es-solución?(sol, k) entonces
            tratar-solución(sol)
        si no
            si es-completable?(sol, k, m) entonces
                vuelta-atrás-marcaje(sol, k + 1, m)
            fsi
        fsi
        m := desmarcar(m, sol[k])
    fmientras
fproc
```

Problemas de optimización

- Se modifican los esquemas anteriores de forma que se almacene la mejor solución encontrada hasta el momento.
- Al tratar una nueva solución, se comparará con la que se tiene almacenada.
- Para realizar esta comparación de la forma más eficiente posible, es preciso almacenar junto con la mejor solución su valor asociado.
- Además, para facilitar el cálculo del valor de cada solución alcanzada se incorpora como marcador el valor (parcial) de la tupla parcial.
- Los problemas de optimización admiten un mecanismo adicional de poda, que se realiza cuando se puede asegurar que ninguno de los descendientes del nodo a expandir puede llegar a alcanzar una solución mejor que la mejor encontrada hasta ese momento.
- En un problema de minimización, una cota inferior o estimación de la mejor solución alcanzable desde un nodo sirve para podarlo si la estimación es ya mayor que el valor asociado a la mejor solución encontrada hasta el momento.

Esquema de optimización

```
proc vuelta-atrás-optimización(sol : tupla, e k : nat, valor : valor,
                                sol-mejor : tupla, valor-mejor : valor)
    preparar-recorrido-nivel(k)
    mientras ¬último-hijo-nivel(k) hacer
        sol[k] := siguiente-hijo-nivel(k)
        valor := actualizar(valor, sol, k)
        si es-solución?(sol, k) entonces
            si mejor(valor, valor-mejor) entonces
                sol-mejor := sol ; valor-mejor := valor
            fsi
        si no
            si es-completable?(sol, k) ∧ es-prometedor?(sol, k, valor, valor-mejor) entonces
                vuelta-atrás-optimización(sol, k + 1, valor, sol-mejor, valor-mejor)
            fsi
        fsi
        valor := desactualizar(valor, sol, k)
    fmientras
fproc
```

Ramificación y poda

- Ramificación y poda es otra técnica de exploración de espacios de soluciones.
- La diferencia esencial con respecto a vuelta atrás es la forma de recorrer el espacio de soluciones.
- Aquí la gestión de los nodos vivos se realiza mediante una cola con prioridad, expandiendo en cada momento el nodo vivo "más prometedor".

Ramificación y poda

- Ramificación y poda es otra técnica de exploración de espacios de soluciones.
- La diferencia esencial con respecto a vuelta atrás es la forma de recorrer el espacio de soluciones.
- Aquí la gestión de los nodos vivos se realiza mediante una cola con prioridad, expandiendo en cada momento el nodo vivo “más prometedor”.
- La variante más utilizada de ramificación y poda es aquella que busca la solución óptima de entre todas las posibles soluciones al problema, esperando encontrarla de forma más rápida que por vuelta atrás.

Ramificación y poda

- Ramificación y poda es otra técnica de exploración de espacios de soluciones.
- La diferencia esencial con respecto a vuelta atrás es la forma de recorrer el espacio de soluciones.
- Aquí la gestión de los nodos vivos se realiza mediante una cola con prioridad, expandiendo en cada momento el nodo vivo "más prometedor".
- La variante más utilizada de ramificación y poda es aquella que busca la solución óptima de entre todas las posibles soluciones al problema, esperando encontrarla de forma más rápida que por vuelta atrás.
- Las características de estos problemas de optimización:
  - la solución es expresable en forma de tupla $(x_1, \ldots, x_n)$,
  - es posible determinar si una tupla es una solución factible,
  - es posible determinar si una tupla parcial $(x_1, \ldots, x_k)$ puede ser completada hasta una solución factible.

Ramificación y poda

- Además, es necesario disponer de una función `coste-estimado` que, dada una tupla parcial $X = (x_1, \ldots, x_k)$, proporcione una cota inferior al coste de la mejor solución alcanzable desde X.
- Si llamamos `coste-real`(X) al coste de dicha mejor solución, entonces ha de cumplirse que

$$\texttt{coste-estimado}(X) \leq \texttt{coste-real}(X).$$

Ramificación y poda

- Además, es necesario disponer de una función `coste-estimado` que, dada una tupla parcial $X = (x_1, \ldots, x_k)$, proporcione una cota inferior al coste de la mejor solución alcanzable desde X.
- Si llamamos `coste-real`(X) al coste de dicha mejor solución, entonces ha de cumplirse que

$$\texttt{coste-estimado}(X) \leq \texttt{coste-real}(X).$$

- Si `coste-estimado`(X) se aproxima suficientemente al valor del coste real, es de esperar que los nodos con un bajo coste estimado conducirán a soluciones con un bajo coste real, y por eso consideraremos como más prometedor aquel nodo vivo que tenga un coste estimado mínimo.

Ramificación y poda

- Además, es necesario disponer de una función `coste-estimado` que, dada una tupla parcial $X = (x_1, \ldots, x_k)$, proporcione una cota inferior al coste de la mejor solución alcanzable desde X.
- Si llamamos $\texttt{coste-real}(X)$ al coste de dicha mejor solución, entonces ha de cumplirse que

$$\texttt{coste-estimado}(X) \leq \texttt{coste-real}(X).$$

- Si $\texttt{coste-estimado}(X)$ se aproxima suficientemente al valor del coste real, es de esperar que los nodos con un bajo coste estimado conducirán a soluciones con un bajo coste real, y por eso consideraremos como más prometedor aquel nodo vivo que tenga un coste estimado mínimo.
- Este coste estimado, como en vuelta atrás, proporciona una forma adicional de poda. Si el coste estimado no es menor que el coste de la mejor solución obtenida hasta el momento, no merece la pena seguir explorando esa rama del árbol.

Esquema general de minimización

```
fun ramificación-y-poda-mín(T : árbol-de-estados) dev ⟨sol-mejor : tupla, coste-mejor : valor⟩
var X, Y : nodo, C : colapr[nodo]
  Y := raíz(T)
  C := cp-vacía() ; añadir(C, Y)
  coste-mejor := +∞
  mientras ¬es-cp-vacía?(C) ∧ coste-estimado(mínimo(C)) < coste-mejor hacer
    Y := mínimo(C) ; eliminar-mín(C)
    para todo hijo X de Y hacer
      si es-solución?(X) entonces
        si coste-real(X) < coste-mejor entonces
          coste-mejor := coste-real(X) ; sol-mejor := solución(X)
        fsi
      si no
        si es-completable?(X) ∧ coste-estimado(X) < coste-mejor entonces
          añadir(C, X)
        fsi
      fsi
    fpara
  fmientras
ffun
```

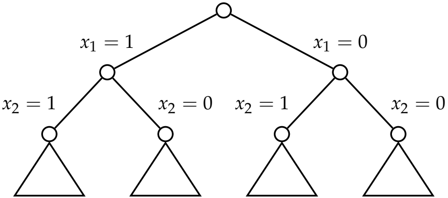
Tareas con plazo fijo, duración y coste

Tenemos n tareas y un procesador. Cada tarea i tiene asociada una terna (t_i, p_i, c_i) que indica que la tarea i tarda un tiempo t_i en ejecutarse, y que si su ejecución no está completa antes del plazo p_i, deberá pagarse una multa c_i por ella.

El objetivo es seleccionar un subconjunto S de las n tareas tal que las tareas en S se puedan realizar antes de su plazo y que la multa impuesta por las tareas que no están en S (no realizadas) sea mínima.

Tareas con plazo fijo, duración y coste

- El subconjunto de tareas realizadas se representa mediante su función característica, es decir, mediante una tupla $(x_1, \ldots, x_n)$ donde $x_i = 1$ indica que la tarea i-ésima se realiza, mientras que $x_i = 0$ indica que no se realiza.
- Se obtiene un árbol de exploración binario.



Tareas con plazo fijo, duración y coste

- Para comprobar la factibilidad del subconjunto seleccionado se puede utilizar la siguiente propiedad: S es factible si y solo si la secuencia que ordena las tareas en S de forma no decreciente según plazo es admisible.

Tareas con plazo fijo, duración y coste

- Para comprobar la factibilidad del subconjunto seleccionado se puede utilizar la siguiente propiedad: S es factible si y solo si la secuencia que ordena las tareas en S de forma no decreciente según plazo es admisible.
- En cuanto a la estimación, necesitamos una cota inferior:
  1. Coste actual, $\sum_{i=1}^{k}(1 - x_i)c_i$.
  2. Coste actual más el coste de las tareas pendientes (desde $k+1$ hasta n) que ya no pueden hacerse antes de su plazo.

Tareas con plazo fijo, duración y coste

tipos
nodo = **reg**
$sol[1..n]$ **de** $0..1$
$k : 0..n$
$tiempo, coste$: *real*
$coste\text{-}estimado$: *real* { prioridad }
freg
ftipos

Tareas con plazo fijo, duración y coste

tipos
nodo = **reg**
$sol[1..n]$ **de** $0..1$
$k : 0..n$
$tiempo, coste$: *real*
$coste\text{-}estimado$: *real* { prioridad }
freg
ftipos

$\{ P[1] \leq P[2] \leq \ldots \leq P[n] \}$
fun `tareas-rp`$(T[1..n], P[1..n], C[1..n]$ **de** $real^{+})$
dev $\langle sol\text{-}mejor[1..n]$ **de** $0..1, coste\text{-}mejor : real\rangle$
var X, Y : *nodo*, $cola$: *colapr[nodo]*
{ generamos la raíz }
$Y.k := 0$; $Y.tiempo := 0$; $Y.coste := 0$
$Y.coste\text{-}estimado :=$ `cálculo-estimación`$(T, P, C, Y.k, Y.tiempo, Y.coste)$
$cola :=$ `cp-vacía`() ; `añadir`$(cola, Y)$
$coste\text{-}mejor := +\infty$

```
mientras ¬es-cp-vacía?(cola) ∧ (mínimo(cola).coste-estimado < coste-mejor) hacer
    Y := mínimo(cola) ; eliminar-mín(cola)
    X.k := Y.k + 1 ;  X.sol := Y.sol
    { hijo izquierdo — ejecutar }
    si Y.tiempo + T[X.k] ≤ P[X.k] entonces      { es factible }
        X.sol[X.k] := 1 ;  X.tiempo := Y.tiempo + T[X.k]
        X.coste := Y.coste
        X.coste-estimado := cálculo-estimación(T, P, C, X.k, X.tiempo, X.coste)
        si X.coste-estimado < coste-mejor entonces
            si X.k = n entonces      { es solución }
                sol-mejor := X.sol ;  coste-mejor := X.coste
            si no
                añadir(cola, X)
            fsi
        fsi
    fsi
```

```
{ hijo derecho — no ejecutar }
X.sol[X.k] := 0 ;  X.tiempo := Y.tiempo
X.coste := Y.coste + C[X.k]
X.coste-estimado := cálculo-estimación(T, P, C, X.k, X.tiempo, X.coste)
si X.coste-estimado < coste-mejor entonces
    si X.k = n entonces
        sol-mejor := X.sol ;  coste-mejor := X.coste
    si no
        añadir(cola, X)
    fsi
fsi
fmientras
fproc
```

```
        { hijo derecho — no ejecutar }
        X.sol[X.k] := 0 ;  X.tiempo := Y.tiempo
        X.coste := Y.coste + C[X.k]
        X.coste-estimado := cálculo-estimación(T, P, C, X.k, X.tiempo, X.coste)
        si X.coste-estimado < coste-mejor entonces
            si X.k = n entonces
                sol-mejor := X.sol ;  coste-mejor := X.coste
            si no
                añadir(cola, X)
            fsi
        fsi
    fmientras
fproc

fun cálculo-estimación(T[1..n], P[1..n], C[1..n] de real⁺, k : 0..n, tiempo, coste : real)
     dev est : real
    est := coste
    para i = k + 1 hasta n hacer
        si tiempo + T[i] > P[i] entonces  est := est + C[i] fsi
    fpara
ffun
```

Esquema optimista/pesimista (minimización)

- En el esquema anterior se mantiene en la variable *coste-mejor* el coste asociado a la mejor solución calculada hasta el momento.
  - Este valor se modifica solamente al encontrar una solución mejor.
  - Al principio tiene el peor valor posible ($+\infty$).
  - La poda no es efectiva hasta encontrar una solución y después mejora lentamente.

Esquema optimista/pesimista (minimización)

- En el esquema anterior se mantiene en la variable *coste-mejor* el coste asociado a la mejor solución calculada hasta el momento.
  - Este valor se modifica solamente al encontrar una solución mejor.
  - Al principio tiene el peor valor posible ($+\infty$).
  - La poda no es efectiva hasta encontrar una solución y después mejora lentamente.
- Suponiendo que disponemos de una función que, dado un nodo X, calcule una cota superior del coste de la mejor solución alcanzable desde X, podría usarse para ir actualizando *coste-mejor*: cada vez que encontremos un nodo factible con una cota superior menor que el valor actual de *coste-mejor*, podremos actualizar *coste-mejor* con el valor de dicha cota, lo que hará la poda más efectiva.

Esquema optimista/pesimista (minimización)

- Usamos, por tanto, dos cotas para cada nodo X:

$$
\begin{array}{lcl}
\texttt{coste-optimista}(X) & = & \text{cota inferior al coste de la mejor solución} \\
 & & \text{alcanzable desde } X \text{ (antes } \texttt{coste-estimado}(X)\text{).} \\
\texttt{coste-pesimista}(X) & = & \text{cota superior del coste de la mejor solución} \\
 & & \text{alcanzable desde } X.
\end{array}
$$

Esquema optimista/pesimista (minimización)

- Usamos, por tanto, dos cotas para cada nodo X:

$$\begin{array}{rcl} \texttt{coste-optimista}(X) & = & \text{cota inferior al coste de la mejor solución} \\ & & \text{alcanzable desde } X \text{ (antes } \texttt{coste-estimado}(X)). \\ \texttt{coste-pesimista}(X) & = & \text{cota superior del coste de la mejor solución} \\ & & \text{alcanzable desde } X. \end{array}$$

Por tanto, dichas cotas cumplen:

$$\texttt{coste-optimista}(X) \leq \texttt{coste-real}(X) \leq \texttt{coste-pesimista}(X).$$

Además para cualquier solución Y alcanzable a partir de X se cumple que

$$\texttt{coste-optimista}(X) \leq \texttt{coste-real}(X) \leq \texttt{coste}(Y).$$

Sin embargo, *no* necesariamente se tiene que cumplir que

$$\texttt{coste}(Y) \leq \texttt{coste-pesimista}(X).$$

Esquema optimista/pesimista (minimización)

fun `rp-opt-pes-mín`(T : *árbol-de-estados*) **dev** $\langle$ *sol-mejor* : *tupla*, *coste-mejor* : *valor* $\rangle$
var X, Y : *nodo*, *cola* : *colapr[nodo]*
    Y := `raíz`(T) ; *cola* := `cp-vacía`() ; `añadir`(*cola*, Y) ; *coste-mejor* := `coste-pesimista`(Y)
    mientras $\neg$`es-cp-vacía?`(*cola*) $\wedge$ `coste-optimista`(`mínimo`(*cola*)) $\leq$ *coste-mejor* **hacer**
        Y := `mínimo`(*cola*) ; `eliminar-mín`(*cola*)
        para todo hijo X de Y **hacer**
            si `es-solución?`(X) **entonces**
                si `coste-real`(X) $\leq$ *coste-mejor* **entonces**
                    coste-mejor := `coste-real`(X) ; *sol-mejor* := X
                fsi
            si no
                si `es-completable?`(X) $\wedge$ `coste-optimista`(X) $\leq$ *coste-mejor* **entonces**
                    `añadir`(*cola*, X)
                    si `coste-pesimista`(X) $<$ *coste-mejor* **entonces**
                        coste-mejor := `coste-pesimista`(X)
                    fsi
                fsi
            fsi
        fpara
    fmientras
ffun

Tareas con plazo fijo, duración y coste

Cota optimista coste actual, $\sum_{i=1}^{k}(1 - x_i)c_i$, más el coste de las tareas pendientes (desde $k + 1$ hasta n) que ya no pueden hacerse antes de su plazo.

Cota pesimista extender a una posible solución.

Tareas con plazo fijo, duración y coste

tipos

nodo = **reg**

$sol[1..n]$ **de** $0..1$

$k : 0..n$

$tiempo, coste$: *real*

$coste\text{-}opt$: *real* { prioridad }

freg

ftipos

Tareas con plazo fijo, duración y coste

tipos
nodo = **reg**
$sol[1..n]$ **de** $0..1$
$k : 0..n$
$tiempo, coste$: *real*
$coste\text{-}opt$: *real* { prioridad }
freg
ftipos

$\{\ P[1] \leq P[2] \leq \ldots \leq P[n]\ \}$
fun `tareas-rp2`$(T[1..n], P[1..n], C[1..n]$ **de** *real*$^+)$
dev $\langle\, sol\text{-}mejor[1..n]$ **de** $0..1, coste\text{-}mejor$: *real*$\,\rangle$
var X, Y : *nodo*, *cola* : *colapr[nodo]*
{ generamos la raíz }
$Y.k := 0$; $Y.tiempo := 0$; $Y.coste := 0$
$Y.coste\text{-}opt :=$ `cálculo-optimista`$(T, P, C, Y.k, Y.tiempo, Y.coste)$
$cola :=$ `cp-vacía`$()$; `añadir`$(cola, Y)$
$coste\text{-}mejor :=$ `cálculo-pesimista`$(T, P, C, Y.k, Y.tiempo, Y.coste)$
mientras $\neg$`es-cp-vacía?`$(cola)$ $\wedge$ (`mínimo`$(cola).coste\text{-}opt \leq coste\text{-}mejor$) **hacer**
$Y :=$ `mínimo`$(cola)$; `eliminar-mín`$(cola)$
$X.k := Y.k + 1$; $X.sol := Y.sol$

{ hijo izquierdo — ejecutar }
si $Y.tiempo + T[X.k] \leq P[X.k]$ **entonces** { es factible }
  $X.sol[X.k] := 1$; $X.tiempo := Y.tiempo + T[X.k]$
  $X.coste := Y.coste$
  $X.coste\text{-}opt :=$ `cálculo-optimista`$(T, P, C, X.k, X.tiempo, X.coste)$
  si $X.coste\text{-}opt \leq coste\text{-}mejor$ **entonces**
    si $X.k = n$ **entonces**
      $sol\text{-}mejor := X.sol$; $coste\text{-}mejor := X.coste$
    si no { $X.k \neq n$ }
      `añadir`$(cola, X)$
      { la estimación pesimista coincide con la de Y }
      { *coste-mejor* no cambia }
    fsi
  fsi
fsi

```
{ hijo derecho — no ejecutar }
X.sol[X.k] := 0 ;  X.tiempo := Y.tiempo
X.coste := Y.coste + C[X.k]
X.coste-opt := cálculo-optimista(T, P, C, X.k, X.tiempo, X.coste)
si X.coste-opt ≤ coste-mejor entonces
    si X.k = n entonces
        sol-mejor := X.sol ;  coste-mejor := X.coste
    si no
        añadir(cola, X)
        pes := cálculo-pesimista(T, P, C, X.k, X.tiempo, X.coste)
        coste-mejor := mín(coste-mejor, pes)
    fsi
fsi
```

```
    fmientras
ffun
```

```
fun cálculo-optimista(T[1..n], P[1..n], C[1..n] de real⁺, k : 0..n, tiempo, coste : real)
     dev opt : real
    opt := coste
    para i = k + 1 hasta n hacer
        si tiempo + T[i] > P[i] entonces
            opt := opt + C[i]
        fsi
    fpara
ffun
```

fun `cálculo-optimista`$(T[1..n], P[1..n], C[1..n]$ **de** $real^+, k : 0..n, tiempo, coste : real)$
  dev $opt : real$
  $opt := coste$
  para $i = k+1$ **hasta** n **hacer**
    si $tiempo + T[i] > P[i]$ **entonces**
      $opt := opt + C[i]$
    fsi
  fpara
ffun

fun `cálculo-pesimista`$(T[1..n], P[1..n], C[1..n]$ **de** $real^+, k : 0..n, tiempo, coste : real)$
  dev $pes : real$
  $pes := coste$; $tiempo\text{-}pes := tiempo$
  para $i = k+1$ **hasta** n **hacer**
    si $tiempo\text{-}pes + T[i] > P[i]$ **entonces**
      $pes := pes + C[i]$
    si no
      $tiempo\text{-}pes := tiempo\text{-}pes + T[i]$
    fsi
  fpara
ffun

Funcionarios con trabajo

El Ministro de Desinformación y Decencia se ha propuesto hacer trabajar en firme a sus n funcionarios, para lo que se ha sacado de la manga n trabajos.

A pesar de su ineficacia, todos los funcionarios son capaces de hacer cualquier trabajo, aunque unos tardan más que otros.

La información al respecto se recoge en la tabla $T[1..n, 1..n]$, donde $T[i, j]$ representa el tiempo que el funcionario i tarda en realizar el trabajo j.

Para justificar su puesto, Su Excelencia el Sr. Ministro desea conocer la asignación óptima de trabajos a funcionarios de modo que la suma total de tiempos sea mínima.

Funcionarios con trabajo

Podemos representar la solución en tuplas de la forma $(x_1, x_2, \ldots, x_n)$ donde x_i es el trabajo asignado al funcionario i.

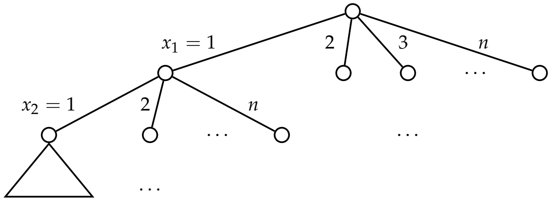
Las soluciones tienen que cumplir que son permutaciones de los n trabajos.

Funcionarios con trabajo

Podemos representar la solución en tuplas de la forma $(x_1, x_2, \ldots, x_n)$ donde x_i es el trabajo asignado al funcionario i.

Las soluciones tienen que cumplir que son permutaciones de los n trabajos.

El árbol de exploración, donde cada nodo tiene n hijos y hay n niveles, es

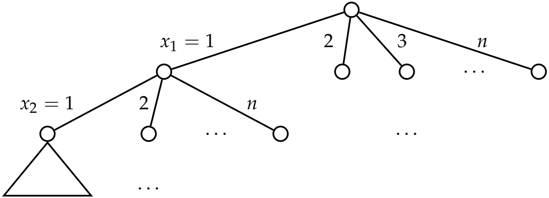


Funcionarios con trabajo

Podemos representar la solución en tuplas de la forma $(x_1, x_2, \ldots, x_n)$ donde x_i es el trabajo asignado al funcionario i.

Las soluciones tienen que cumplir que son permutaciones de los n trabajos.

El árbol de exploración, donde cada nodo tiene n hijos y hay n niveles, es



Ya que cada trabajo solo puede ser asignado a un funcionario, llevaremos cuenta de los trabajos ya asignados en un vector marcador $asignado[1..n]$ de booleanos.

Elegir los x_i tales que se minimice $\sum_{i=1}^{n} T[i, x_i]$.

Funcionarios con trabajo: cota optimista

Veamos cómo podemos obtener una cota inferior del coste total a partir del coste de una solución parcial.

Para la solución parcial $(x_1, \ldots, x_k)$, el tiempo hasta el momento es $\mathit{tiempo} = \sum_{i=1}^{k} T[i, x_i]$, y tenemos que estimar el tiempo del resto de la solución.

Funcionarios con trabajo: cota optimista

Veamos cómo podemos obtener una cota inferior del coste total a partir del coste de una solución parcial.

Para la solución parcial $(x_1, \ldots, x_k)$, el tiempo hasta el momento es $\mathit{tiempo} = \sum_{i=1}^{k} T[i, x_i]$, y tenemos que estimar el tiempo del resto de la solución.

1. La opción más sencilla (y más optimista) es aproximar con 0 este tiempo, y utilizar *tiempo* como estimación.

Funcionarios con trabajo: cota optimista

Veamos cómo podemos obtener una cota inferior del coste total a partir del coste de una solución parcial.

Para la solución parcial $(x_1, \ldots, x_k)$, el tiempo hasta el momento es $tiempo = \sum_{i=1}^{k} T[i, x_i]$, y tenemos que estimar el tiempo del resto de la solución.

1. La opción más sencilla (y más optimista) es aproximar con 0 este tiempo, y utilizar *tiempo* como estimación.
2. Otra posibilidad es calcular un mínimo global de la matriz T,

$$mínT = \text{mín}\{T[i,j] \mid 1 \leq i \leq n \ \wedge \ 1 \leq j \leq n\},$$

   que sirva como cota inferior al tiempo de realización de cada trabajo por los funcionarios.
   Así podemos aproximar el tiempo del resto de la solución con $(n-k)\,mínT$.

Funcionarios con trabajo: cota optimista

3. Tener calculado un mínimo por cada fila: para cada funcionario, cuánto tarda en realizar el trabajo que realiza más rápidamente:

$$rápido[i] = \min\{T[i,j] \mid 1 \leq j \leq n\}.$$

De esta manera podemos calcular la estimación del tiempo restante como:

$$\sum_{i=k+1}^{n} rápido[i].$$

Funcionarios con trabajo: cota optimista

3. Tener calculado un mínimo por cada fila: para cada funcionario, cuánto tarda en realizar el trabajo que realiza más rápidamente:

$$rápido[i] = \text{mín}\{T[i,j] \mid 1 \leq j \leq n\}.$$

De esta manera podemos calcular la estimación del tiempo restante como:

$$\sum_{i=k+1}^{n} rápido[i].$$

4. Calcular *rápido* dinámicamente entre los trabajos no repartidos ya:

$$rápido\text{-}din[i] = \text{mín}\{T[i,j] \mid 1 \leq j \leq n \ \wedge \ j \text{ no ha sido asignado}\}.$$

Funcionarios con trabajo: cota optimista

3. Tener calculado un mínimo por cada fila: para cada funcionario, cuánto tarda en realizar el trabajo que realiza más rápidamente:

$$\textit{rápido}[i] = \text{mín}\{T[i,j] \mid 1 \leq j \leq n\}.$$

   De esta manera podemos calcular la estimación del tiempo restante como:

$$\sum_{i=k+1}^{n} \textit{rápido}[i].$$

4. Calcular *rápido* dinámicamente entre los trabajos no repartidos ya:

$$\textit{rápido-din}[i] = \text{mín}\{T[i,j] \mid 1 \leq j \leq n \ \wedge \ j \text{ no ha sido asignado}\}.$$

Implementamos la opción 3. Además calcularemos inicialmente las sumas

$$\textit{est}[k] = \sum_{i=k+1}^{n} \textit{rápido}[i]$$

de tal forma que no tengan que ser recalculados dentro del algoritmo.

Funcionarios con trabajo: cota pesimista

1. Derivar una solución cualquiera a partir de la solución parcial, asignando trabajos libres a los funcionarios restantes siguiendo el orden establecido.

Funcionarios con trabajo: cota pesimista

1. Derivar una solución cualquiera a partir de la solución parcial, asignando trabajos libres a los funcionarios restantes siguiendo el orden establecido.
2. También se puede obtener una cota superior calculando el máximo global de la matriz T,

$$\mathit{máxT} = \text{máx}\{T[i,j] \mid 1 \leq i \leq n \ \wedge \ 1 \leq j \leq n\},$$

   y aproximando superiormente el tiempo del resto de la solución con $(n-k)\,\mathit{máxT}$.

Funcionarios con trabajo: cota pesimista

1. Derivar una solución cualquiera a partir de la solución parcial, asignando trabajos libres a los funcionarios restantes siguiendo el orden establecido.
2. También se puede obtener una cota superior calculando el máximo global de la matriz T,

$$\mathit{máxT} = \max\{T[i,j] \mid 1 \leq i \leq n \ \wedge \ 1 \leq j \leq n\},$$

   y aproximando superiormente el tiempo del resto de la solución con $(n-k)\,\mathit{máxT}$.
3. Esta cota se puede mejorar calculando un máximo por cada fila:

$$\mathit{lento}[i] = \max\{T[i,j] \mid 1 \leq j \leq n\}.$$

   De esta manera podemos precalcular la estimación del tiempo restante como

$$\mathit{pes}[k] = \sum_{i=k+1}^{n} \mathit{lento}[i].$$

Funcionarios con trabajo: cota pesimista

1. Derivar una solución cualquiera a partir de la solución parcial, asignando trabajos libres a los funcionarios restantes siguiendo el orden establecido.
2. También se puede obtener una cota superior calculando el máximo global de la matriz T,

$$\mathit{máxT} = \text{máx}\{T[i,j] \mid 1 \leq i \leq n \ \wedge \ 1 \leq j \leq n\},$$

   y aproximando superiormente el tiempo del resto de la solución con $(n-k)\,\mathit{máxT}$.
3. Esta cota se puede mejorar calculando un máximo por cada fila:

$$\mathit{lento}[i] = \text{máx}\{T[i,j] \mid 1 \leq j \leq n\}.$$

   De esta manera podemos precalcular la estimación del tiempo restante como

$$\mathit{pes}[k] = \sum_{i=k+1}^{n} \mathit{lento}[i].$$

4. Podríamos afinar la estimación, calculando *lento* dinámicamente:

$$\mathit{lento\text{-}din}[i] = \text{máx}\{T[i,j] \mid 1 \leq j \leq n \ \wedge \ j \text{ no ha sido asignado}\}.$$

Funcionarios con trabajo: implementación

tipos

$$
\begin{array}{lll}
\textit{nodo} & = & \textbf{reg} \\
 & & \quad sol[1..n]\ \textbf{de}\ 1..n \\
 & & \quad k : 0..n \\
 & & \quad tiempo : \textit{real} \\
 & & \quad tiempo\text{-}opt : \textit{real} \qquad \{ \text{ prioridad } \} \\
 & & \quad asignado[1..n]\ \textbf{de}\ \textit{bool} \\
 & & \textbf{freg}
\end{array}
$$

ftipos

Funcionarios con trabajo: implementación

tipos
    nodo = **reg**
        $sol[1..n]$ **de** $1..n$
        $k : 0..n$
        $tiempo$: *real*
        $tiempo\text{-}opt$: *real*   { prioridad }
        $asignado[1..n]$ **de** *bool*
    freg
ftipos

fun `funcionarios-mín-rp`$(T[1..n, 1..n]$ **de** $real^+)$
  dev $\langle sol\text{-}mejor[1..n]$ **de** $1..n, tiempo\text{-}mejor$: *real*$\rangle$
var X, Y : *nodo*, C : *colapr[nodo]*, $opt[0..n], pes[0..n]$ **de** *real*
  $\langle opt, pes \rangle :=$ `pre-cálculo-estimaciones`(T)
  { generamos la raíz }
  $Y.k := 0$; $Y.asignado[1..n] := [$`falso`$]$
  $Y.tiempo := 0$; $Y.tiempo\text{-}opt := opt[0]$
  $C :=$ `cp-vacía`() ; `añadir`(C, Y)
  $tiempo\text{-}mejor := pes[0]$

Funcionarios con trabajo: implementación

```
mientras ¬es-cp-vacía?(C) ∧ mínimo(C).tiempo-opt ≤ tiempo-mejor hacer
    Y := mínimo(C) ; eliminar-mín(C)
    { generamos los hijos de Y }
    X.k := Y.k + 1 ; X.sol := Y.sol ; X.asignado := Y.asignado
    para t = 1 hasta n hacer
        si ¬X.asignado[t] entonces
            X.sol[X.k] := t ; X.asignado[t] := cierto
            X.tiempo := Y.tiempo + T[X.k, t]
            X.tiempo-opt := X.tiempo + opt[X.k]
            si X.tiempo-opt ≤ tiempo-mejor entonces
                si X.k = n entonces
                    sol-mejor := X.sol ; tiempo-mejor := X.tiempo
                si no
                    añadir(C, X)
                    tiempo-mejor := mín(tiempo-mejor, X.tiempo + pes[X.k])
                fsi
            fsi
            X.asignado[t] := falso
        fsi
    fpara
fmientras
ffun
```

Funcionarios con trabajo: implementación

fun `pre-cálculo-estimaciones`$(T[1..n,1..n]$ **de** *real*$^+)$ **dev** $\langle opt[0..n], pes[0..n]$ **de** *real*$\rangle$
var $rápido[1..n], lento[1..n]$ **de** *real*
  { cálculo de los mínimos y máximos por filas }
  para $i = 1$ **hasta** n **hacer**
    $rápido[i] := T[i,1]$
    $lento[i] := T[i,1]$
    para $j = 2$ **hasta** n **hacer**
      $rápido[i] :=$ `mín`$(rápido[i], T[i,j])$
      $lento[i] :=$ `máx`$(lento[i], T[i,j])$
    fpara
  fpara
  { cálculo de las estimaciones }
  $opt[n] := 0$; $pes[n] := 0$
  para $i = n-1$ **hasta** 0 **paso** -1 **hacer**
    $opt[i] := opt[i+1] + rápido[i+1]$
    $pes[i] := pes[i+1] + lento[i+1]$
  fpara
ffun

$$\{\ opt[k] = \sum_{i=k+1}^{n} \text{mín}_{1\leq j\leq n}\{T[i,j]\} \wedge pes[k] = \sum_{i=k+1}^{n} \text{máx}_{1\leq j\leq n}\{T[i,j]\}\ \}$$

Problema de la mochila

Cuando Alí-Babá consigue por fin entrar en la Cueva de los Cuarenta Ladrones encuentra allí gran cantidad de objetos muy valiosos. A pesar de su pobreza, Alí-Babá conoce muy bien el peso y valor de cada uno de los objetos en la cueva.

Debido a los peligros que tiene que afrontar en su camino de vuelta, solo puede llevar consigo aquellas riquezas que quepan en su pequeña mochila, que soporta un peso máximo conocido.

Determinar qué objetos debe elegir Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila.

Problema de la mochila

Hay n objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$ para todo i entre 1 y n.

La mochila soporta un peso total máximo $M > 0$.

Problema de la mochila

Hay n objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$ para todo i entre 1 y n.

La mochila soporta un peso total máximo $M > 0$.

El problema consiste en maximizar

$$\sum_{i=1}^{n} x_i v_i$$

con la restricción

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

donde $x_i \in \{0, 1\}$ indica si hemos cogido (1) o no (0) el objeto i.

Problema de la mochila

Hay n objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$ para todo i entre 1 y n.

La mochila soporta un peso total máximo $M > 0$.

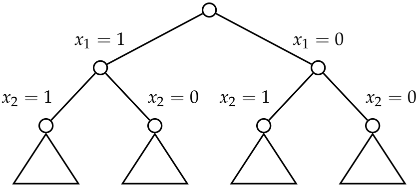
El problema consiste en maximizar

$$\sum_{i=1}^{n} x_i v_i$$

con la restricción

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

donde $x_i \in \{0, 1\}$ indica si hemos cogido (1) o no (0) el objeto i.



Esquema optimista/pesimista (maximización)

Hay que modificar ligeramente los esquemas vistos. En concreto, la cola con prioridad será de máximos, y se cumplirá

$$\texttt{benef-optimista}(X) \geq \texttt{benef-real}(X) \geq \texttt{benef-pesimista}(X).$$

Esquema optimista/pesimista (maximización)

Hay que modificar ligeramente los esquemas vistos. En concreto, la cola con prioridad será de máximos, y se cumplirá

$$\texttt{benef-optimista}(X) \geq \texttt{benef-real}(X) \geq \texttt{benef-pesimista}(X).$$

fun $\texttt{rp-opt-pes-máx}(T : \textit{árbol-de-estados})$ **dev** $\langle \textit{sol-mejor} : \textit{tupla}, \textit{benef-mejor} : \textit{valor} \rangle$
var X, Y : *nodo*, *cola* : *colapr[nodo]*
    $Y := \texttt{raíz}(T)$; $\textit{cola} := \texttt{cp-vacía}()$; $\texttt{añadir}(\textit{cola}, Y)$; $\textit{benef-mejor} := \texttt{benef-pesimista}(Y)$
    mientras $\neg\texttt{es-cp-vacía?}(\textit{cola}) \wedge \texttt{benef-optimista}(\texttt{máximo}(\textit{cola})) \geq \textit{benef-mejor}$ **hacer**
        $Y := \texttt{máximo}(\textit{cola})$; $\texttt{eliminar-máx}(\textit{cola})$
        para todo hijo X de Y **hacer**
            si $\texttt{es-solución?}(X)$ **entonces**
                si $\texttt{benef-real}(X) \geq \textit{benef-mejor}$ **entonces**
                    $\textit{benef-mejor} := \texttt{benef-real}(X)$; $\textit{sol-mejor} := X$
                fsi
            si no
                si $\texttt{es-completable?}(X) \wedge \texttt{benef-optimista}(X) \geq \textit{benef-mejor}$ **entonces**
                    $\texttt{añadir}(\textit{cola}, X)$
                    si $\texttt{benef-pesimista}(X) > \textit{benef-mejor}$ **entonces**
                        $\textit{benef-mejor} := \texttt{benef-pesimista}(X)$
            fsi fsi fsi
        fpara
    fmientras
ffun

Problema de la mochila: cota optimista

- Como el problema es de maximización, tenemos que encontrar una cota superior de la mejor solución alcanzable.

Problema de la mochila: cota optimista

- Como el problema es de maximización, tenemos que encontrar una cota superior de la mejor solución alcanzable.
- Utilizamos el algoritmo voraz que resolvía este mismo problema cuando los objetos se podían fraccionar ($0 \leq x_i \leq 1$).
- Ya que esta solución era óptima, no puede haber ninguna solución sin fraccionar objetos que sea mejor.
- Para utilizar este algoritmo necesitamos que los objetos estén ordenados en orden decreciente de valor por unidad de peso, v_i/p_i.

Problema de la mochila: cota pesimista

- Podríamos tomar como cota inferior el valor de los objetos que ya se han cogido.
- Una cota que se aproxima más al beneficio real consiste en probar una posible solución: de entre los objetos sobre los cuales todavía no se ha decidido, se incorporan a la mochila todos los objetos que se pueda, considerándolos en el orden establecido.
- Cuando nos encontramos en un nodo en el que el último objeto considerado se ha metido en la mochila, la cota pesimista coincide con la de su padre, por lo que no se podrá mejorar la variable *beneficio-mejor*.

Problema de la mochila: implementación

tipos

nodo $=$ **reg**

$sol[1..n]$ **de** $0..1$

$k : 0..n$

$peso, beneficio$: *real*

$beneficio\text{-}opt$: *real* { prioridad }

freg

ftipos

Problema de la mochila: implementación

tipos
nodo = **reg**
$sol[1..n]$ **de** $0..1$
$k : 0..n$
$peso, beneficio$: *real*
$beneficio\text{-}opt$: *real* { prioridad }
freg
ftipos

$\{ \frac{V[1]}{P[1]} \geq \frac{V[2]}{P[2]} \geq \ldots \geq \frac{V[n]}{P[n]} \}$
fun `mochila-rp`$(P[1..n], V[1..n]$ **de** *real*$^+, M :$ *real*$^+)$
dev $\langle sol\text{-}mejor[1..n]$ **de** $0..1, beneficio\text{-}mejor :$ *real*$\rangle$
var X, Y : *nodo*, C : *colapr[nodo]*
{ generamos la raíz }
$Y.k := 0\ ;\ Y.peso := 0\ ;\ Y.beneficio := 0$
$\langle Y.beneficio\text{-}opt, beneficio\text{-}mejor \rangle :=$ `cálculo-estimaciones`$(P, V, M, Y.k, Y.peso, Y.beneficio)$
$C :=$ `cp-vacía`$()$; `añadir`(C, Y)
mientras $\neg$`es-cp-vacía?`$(C) \wedge$ `máximo`$(C).beneficio\text{-}opt \geq beneficio\text{-}mejor$ **hacer**
$Y :=$ `máximo`(C) ; `eliminar-máx`(C)
$X.k := Y.k + 1\ ;\ X.sol := Y.sol$

Problema de la mochila: implementación

```
        { probamos a meter el objeto en la mochila }
        si Y.peso + P[X.k] ≤ M entonces
            { es factible y, por tanto, las estimaciones coinciden con las de Y }
            { beneficio-opt(X) = beneficio-opt(Y) ≥ beneficio-mejor }
            X.sol[X.k] := 1 ; X.peso := Y.peso + P[X.k]
            X.beneficio := Y.beneficio + V[X.k] ; X.beneficio-opt := Y.beneficio-opt
            si X.k = n entonces { beneficio(X) = beneficio-opt(X) ≥ beneficio-mejor }
                sol-mejor := X.sol ; beneficio-mejor := X.beneficio
            si no añadir(C, X)    { no se puede mejorar beneficio-mejor }
            fsi
        fsi
        { probamos a no meter el objeto (siempre es factible) }
        ⟨X.beneficio-opt, pes⟩ := cálculo-estimaciones(P, V, M, X.k, Y.peso, Y.beneficio)
        si X.beneficio-opt ≥ beneficio-mejor entonces
            X.sol[X.k] := 0 ; X.peso := Y.peso ; X.beneficio := Y.beneficio
            si X.k = n entonces
                sol-mejor := X.sol ; beneficio-mejor := X.beneficio
            si no
                añadir(C, X) ; beneficio-mejor := máx(beneficio-mejor, pes)
            fsi
        fsi
    fmientras
ffun
```

Problema de la mochila: implementación

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

fun `cálculo-estimaciones`$(P[1..n], V[1..n]$ **de** *real*$^{+}$, M : *real*$^{+}$, $k : 0..n$, *peso*, *beneficio* : *real*$)$
  dev $\langle$ *opt*, *pes* : *real* $\rangle$
  $hueco := M - peso$; $pes := beneficio$; $opt := beneficio$
  $j := k+1$
  mientras $j \leq n \wedge P[j] \leq hueco$ **hacer**
    { podemos coger el objeto j entero }
    $hueco := hueco - P[j]$; $opt := opt + V[j]$; $pes := pes + V[j]$
    $j := j+1$
  fmientras
  si $j \leq n$ **entonces** { quedan objetos por probar y $P[j] > hueco$ }
    { fraccionamos el objeto j (solución voraz) }
    $opt := opt + (hueco/P[j]) * V[j]$
    { extendemos a una solución en la versión 0/1 }
    $j := j+1$
    mientras $j \leq n \wedge hueco > 0$ **hacer**
      si $P[j] \leq hueco$ **entonces**
        $hueco := hueco - P[j]$; $pes := pes + V[j]$
      fsi
      $j := j+1$
    fmientras
  fsi
ffun

Problema del viajante

Un vendedor ambulante de alfombras persas tiene que recorrer n ciudades volviendo tras ello al punto de partida. El buen señor se ha informado sobre las posibles conexiones directas por ferrocarril entre las ciudades y sobre las tarifas correspondientes.

El vendedor desea conocer un circuito en tren que recorra cada ciudad exactamente una vez y regrese a la ciudad de partida, y cuya tarifa total sea mínima.

Problema del viajante

Representamos las soluciones como tuplas $(x_1, \ldots, x_n)$, donde x_i es el vértice por el que pasamos en i-ésimo lugar.

Problema del viajante

Representamos las soluciones como tuplas $(x_1, \ldots, x_n)$, donde x_i es el vértice por el que pasamos en i-ésimo lugar.

Las soluciones tienen que cumplir que se utilizan vértices válidos, que no hay repeticiones y que siempre hay arista de uno al siguiente, y que hay arista del último al primero.

Problema del viajante

Representamos las soluciones como tuplas $(x_1, \ldots, x_n)$, donde x_i es el vértice por el que pasamos en i-ésimo lugar.

Las soluciones tienen que cumplir que se utilizan vértices válidos, que no hay repeticiones y que siempre hay arista de uno al siguiente, y que hay arista del último al primero.

Para evitar soluciones repetidas fijamos el comienzo de los ciclos en el vértice 1, es decir, $x_1 = 1$, y empezaremos a tomar decisiones a partir del segundo vértice del ciclo.

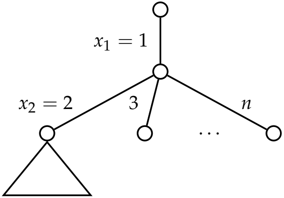
Problema del viajante

Representamos las soluciones como tuplas $(x_1, \ldots, x_n)$, donde x_i es el vértice por el que pasamos en i-ésimo lugar.

Las soluciones tienen que cumplir que se utilizan vértices válidos, que no hay repeticiones y que siempre hay arista de uno al siguiente, y que hay arista del último al primero.

Para evitar soluciones repetidas fijamos el comienzo de los ciclos en el vértice 1, es decir, $x_1 = 1$, y empezaremos a tomar decisiones a partir del segundo vértice del ciclo.

En el árbol de exploración cada nodo excepto la raíz tiene $n-1$ hijos, y hay n niveles.



Problema del viajante: cotas

El coste de las soluciones será de la forma

$$\underbrace{\sum_{i=2}^{k} \texttt{gv-valor}(x_{i-1}, x_i, G)}_{\text{fijo}} + \underbrace{\left(\sum_{i=k+1}^{n} \texttt{gv-valor}(x_{i-1}, x_i, G) \right) + \texttt{gv-valor}(x_n, x_1, G)}_{n-k+1 \text{ aristas}}$$

por lo que podemos aproximar el coste de las últimas $n-k+1$ aristas con $(n-k+1)\, mínG$, donde $mínG$ es el valor mínimo de todas las aristas de G.

Problema del viajante: cotas

El coste de las soluciones será de la forma

$$\underbrace{\sum_{i=2}^{k} \texttt{gv-valor}(x_{i-1}, x_i, G)}_{\text{fijo}} + \underbrace{\left(\sum_{i=k+1}^{n} \texttt{gv-valor}(x_{i-1}, x_i, G)\right) + \texttt{gv-valor}(x_n, x_1, G)}_{n-k+1 \text{ aristas}}$$

por lo que podemos aproximar el coste de las últimas $n-k+1$ aristas con $(n-k+1)\, mínG$, donde $mínG$ es el valor mínimo de todas las aristas de G.

Este problema puede no tener solución, lo que representamos con un coste igual a $+\infty$.

No podemos utilizar una cota superior del coste de una solución parcial, porque no se garantiza que exista alguna solución en el correspondiente subárbol.

No podemos utilizar una cota pesimista, y *coste-mejor* se actualizará solo al encontrar una solución.

Problema del viajante: implementación

tipos

$$
\begin{array}{lll}
\mathit{nodo} & = & \textbf{reg} \\
 & & \quad \mathit{sol}[1..n] \ \textbf{de} \ 1..n \\
 & & \quad k : 1..n \\
 & & \quad \mathit{coste} : \mathit{real} \\
 & & \quad \mathit{coste\text{-}estimado} : \mathit{real} \qquad \{ \text{ prioridad } \} \\
 & & \quad \mathit{usado}[1..n] \ \textbf{de} \ \mathit{bool} \\
 & & \textbf{freg}
\end{array}
$$

ftipos

Problema del viajante: implementación

tipos
  nodo = **reg**
    $sol[1..n]$ **de** $1..n$
    $k : 1..n$
    $coste$: *real*
    $coste\text{-}estimado$: *real*   { prioridad }
    $usado[1..n]$ **de** *bool*
  freg
ftipos

fun `vendedor-rp`$(G :$ *grafo-val[n]*$)$ **dev** $\langle sol\text{-}mejor[1..n]$ **de** $1..n, coste\text{-}mejor : real_\infty \rangle$
var X, Y : *nodo*, C : *colapr[nodo]*
  $mínG$:= `cálculo-mínimo`(G)
  { generamos la raíz }
  $Y.k := 1$; $Y.sol[1] := 1$; $Y.usado[1] :=$ `cierto` ; $Y.usado[2..n] :=$ [`falso`]
  $Y.coste := 0$; $Y.coste\text{-}estimado := n * mínG$
  $C :=$ `cp-vacía`() ; `añadir`(C, Y)
  $coste\text{-}mejor := +\infty$ { para indicar que no hay solución }
  mientras $\neg$`es-cp-vacía?`$(C) \wedge$ `mínimo`$(C).coste\text{-}estimado < coste\text{-}mejor$ **hacer**
    $Y :=$ `mínimo`(C) ; `eliminar-mín`(C)
    { generamos los hijos de Y }
    $X.k := Y.k + 1$; $X.sol := Y.sol$; $X.usado := Y.usado$
    $anterior := X.sol[X.k - 1]$

Problema del viajante: implementación

```
        para vértice = 2 hasta n hacer
            si ¬X.usado[vértice] ∧ gv-está-arista?(anterior, vértice, G) entonces
                X.sol[X.k] := vértice ; X.usado[vértice] := cierto
                X.coste := Y.coste + gv-valor(anterior, vértice, G)
                si X.k = n entonces
                    si gv-está-arista?(sol[n], 1, G) ∧c
                        X.coste + gv-valor(X.sol[n], 1, G) < coste-mejor entonces
                            sol-mejor := X.sol ; coste-mejor := X.coste + gv-valor(X.sol[n], 1, G)
                    fsi
                si no
                    X.coste-estimado := X.coste + (n − X.k + 1) ∗ mínG
                    si X.coste-estimado < coste-mejor entonces
                        añadir(C, X)
                    fsi
                fsi
                X.usado[vértice] := falso
            fsi
        fpara
    fmientras
ffun
```

Envasado de objetos

Se tiene una colección de n objetos "moldeables", cada uno con un volumen v_i, para i entre 1 y n, que hay que empaquetar utilizando envases de capacidad E. Queremos conocer el empaquetamiento óptimo, es decir, que minimice la cantidad de envases utilizados, teniendo en cuenta que los objetos no se pueden fraccionar.

Envasado de objetos

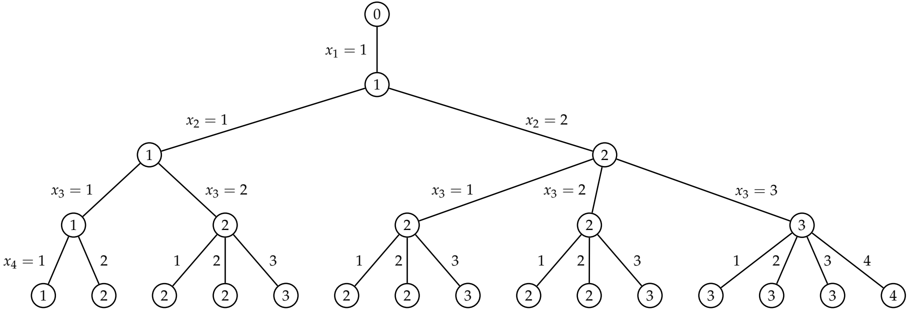
Se tiene una colección de n objetos "moldeables", cada uno con un volumen v_i, para i entre 1 y n, que hay que empaquetar utilizando envases de capacidad E. Queremos conocer el empaquetamiento óptimo, es decir, que minimice la cantidad de envases utilizados, teniendo en cuenta que los objetos no se pueden fraccionar.

Supondremos que todo objeto cabe en un envase vacío. Ya que necesitamos como mucho n envases, podemos numerar los envases del 1 al n.

Las soluciones podemos representarlas en tuplas de la forma $(x_1, \ldots, x_n)$, siendo x_i el envase donde hemos colocado el objeto i. Como todos los envases vacíos son iguales, para cada objeto se puede usar uno de los envases ya ocupados, si cabe en alguno, o coger uno cualquiera sin usar (vacío). El primer objeto siempre se colocará en el primer envase ($x_1 = 1$).

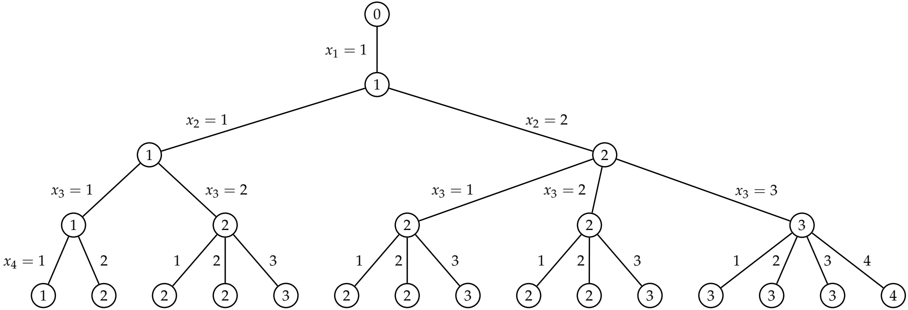
Envasado de objetos: árbol de exploración

Para $n = 4$:



Envasado de objetos: árbol de exploración

Para $n = 4$:



Ya que para colocar un objeto en un envase tenemos que disponer de suficiente espacio disponible en el envase, llevaremos cuenta de la capacidad libre en cada envase mediante un vector *capacidad*$[1..n]$.

Envasado de objetos: cota optimista

El número de *envases* ya utilizados en una solución parcial es una posible estimación del número de envases necesarios al extender dicha solución.

Envasado de objetos: cota optimista

El número de *envases* ya utilizados en una solución parcial es una posible estimación del número de envases necesarios al extender dicha solución.

Otra estimación correcta es considerar que los objetos restantes son fragmentables, e intentar empaquetarlos todos comenzando por rellenar el volumen disponible en los envases ya utilizados:

$$\mathit{envases\text{-}estimados} = \mathit{envases} + \text{máx}\left\{0, \left\lceil \frac{\mathit{pendiente} - \mathit{hueco}}{E} \right\rceil\right\}$$

donde
$\mathit{pendiente}$ = cantidad del total de volumen que queda por empaquetar
hueco = suma de todas las capacidades restantes en los envases ya utilizados

Envasado de objetos: cota optimista

El número de *envases* ya utilizados en una solución parcial es una posible estimación del número de envases necesarios al extender dicha solución.

Otra estimación correcta es considerar que los objetos restantes son fragmentables, e intentar empaquetarlos todos comenzando por rellenar el volumen disponible en los envases ya utilizados:

$$\mathit{envases\text{-}estimados} = \mathit{envases} + \text{máx}\left\{0, \left\lceil \frac{\mathit{pendiente} - \mathit{hueco}}{E} \right\rceil\right\}$$

donde
pendiente = cantidad del total de volumen que queda por empaquetar
hueco = suma de todas las capacidades restantes en los envases ya utilizados

Esta segunda estimación nunca poda más. Si es mayor que *envases* es igual a un valor que solamente depende de todos los volúmenes v_i, y no del reparto ya hecho en la solución parcial:

$$\mathit{óptimo} = \left\lceil \frac{\sum_{i=1}^{n} v_i}{E} \right\rceil,$$

y es igual al número de envases en una solución óptima. Se puede utilizar para parar la búsqueda.

Envasado de objetos: cota pesimista

Una cota superior muy sencilla es considerar un envase extra por cada objeto que nos queda por empaquetar, pero resulta demasiado grosera.

Podemos en cambio ir considerando cada objeto restante, en el orden que se haya dado, e intentar meterlo en el primer envase utilizado y, en el caso en que no quepa, intentarlo con el segundo envase, y así hasta agotar todas las posibilidades, en cuyo caso, se añadirá un nuevo envase a la solución parcial.

Envasado de objetos

tipos

nodo = **reg**

$sol[1..n]$ **de** $1..n$

$k : 1..n$

$envases : 1..n$ { prioridad }

$capacidad[1..n]$ **de** *real*

freg

ftipos

$\{\ \forall i : 1 \le i \le n : V[i] \le E\ \}$

fun `empaquetar-rp`$(E : real^+, V[1..n]$ **de** $real^+)$ **dev** $\langle\, sol\text{-}mejor[1..n]$ **de** $1..n, envases\text{-}mejor : 1..n\,\rangle$

var X, Y : *nodo*, C : *colapr[nodo]*

$total := 0$

para $i = 1$ **hasta** n **hacer** $total := total + V[i]$ **fpara**

$óptimo := \lceil total/E \rceil$

$encontrada :=$ `falso`

{ generamos la raíz: el primer objeto en el primer envase }

$Y.k := 1\ ;\ Y.sol[1] := 1\ ;\ Y.envases := 1$

$Y.capacidad[1] := E - V[1]\ ;\ Y.capacidad[2..n] := [E]$

$C :=$ `cp-vacía`$()$; `añadir`(C, Y)

$envases\text{-}mejor :=$ `cálculo-pesimista`(E, V, Y)

Envasado de objetos

```
mientras ¬encontrada ∧ ¬es-cp-vacía?(C) ∧ mínimo(C).envases ≤ envases-mejor hacer
    Y := mínimo(C) ; eliminar-mín(C)
    { generamos los hijos de Y }
    X.k := Y.k + 1 ;  X.sol := Y.sol
    X.envases := Y.envases ;  X.capacidad := Y.capacidad
    { probamos con cada envase ya utilizado }
    i := 1
    mientras i ≤ Y.envases ∧ ¬encontrada hacer
        si X.capacidad[i] ≥ V[X.k] entonces
            X.sol[X.k] := i ;  X.capacidad[i] := X.capacidad[i] − V[X.k]
            si X.k = n entonces
                sol-mejor := X.sol ;  envases-mejor := X.envases
                encontrada := (envases-mejor = óptimo)      { terminar }
            si no
                añadir(C, X)
                pes := cálculo-pesimista(E, V, X)
                envases-mejor := mín(envases-mejor, pes)
            fsi
            X.capacidad[i] := Y.capacidad[i]
        fsi
        i := i + 1
    fmientras
```

Envasado de objetos

```
si ¬encontrada entonces
    { probamos con un envase nuevo }
    nuevo := Y.envases + 1
    X.sol[X.k] := nuevo ;  X.envases := nuevo
    X.capacidad[nuevo] := E − V[X.k]
    si X.envases ≤ envases-mejor entonces
        si X.k = n entonces
            sol-mejor := X.sol ;  envases-mejor := nuevo
            encontrada := (envases-mejor = óptimo)      { terminar }
        si no
            añadir(C, X)
            pes := cálculo-pesimista(E, V, X)
            envases-mejor := mín(envases-mejor, pes)
        fsi
    fsi
fsi
fmientras
ffun
```

Envasado de objetos

fun `cálculo-pesimista`$(E : real^+, V[1..n]$ **de** $real^+, X : nodo)$ **dev** $pes : 1..n$
var $capacidad\text{-}aux[1..n]$ **de** $real$
    $pes := X.envases$; $capacidad\text{-}aux := X.capacidad$
    para $i = X.k+1$ **hasta** n **hacer**
        $j := 1$
        mientras $V[i] > capacidad\text{-}aux[j]$ **hacer** $j := j+1$ **fmientras**
        $capacidad\text{-}aux[j] := capacidad\text{-}aux[j] - V[i]$
        $pes :=$ `max`(pes, j)
    fpara
ffun