UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Algorítmica (11593)

Voraces y RP–08 de enero de 2018

Escuela Técnica Superior de Ingeniería Informática

etsinf

NOMBRE: _____ **NÚM.:** _____

1 **4 puntos**

Dada una mochila de capacidad W y una serie de N objetos de peso w_i para i entre 1 y N, se desea **llenar completamente** la mochila maximizando el número de objetos que se introducen. Para ello:

a) Expresa el problema en términos de optimización: expresa formalmente el conjunto de soluciones factibles, la función objetivo a maximizar y la solución óptima buscada.
b) Describe los siguientes conceptos sobre los estados que serán necesarios para el algoritmo:
   1) Representación de un estado (no terminal) y su coste espacial.
   2) Condición para que un estado sea solución.
   3) Identifica el estado inicial que representa todo el conjunto de soluciones factibles.
c) Define una función de ramificación. Contesta a las siguientes cuestiones:
   1) Explica la función.
   2) Define la función (en python o en lenguaje matemático). Calcula el coste temporal.
d) Diseña una cota optimista no trivial. Contesta a las siguientes cuestiones:
   1) Explica la cota (caso general, de un estado intermedio).
   2) Explica la cota del estado inicial.
   3) Define la cota (en python o en lenguaje matemático). Calcula el coste temporal.
   4) Estudia si se puede mejorar el cálculo de la cota, haciéndolo incremental. Define la cota así definida (en python o en lenguaje matemático). Calcula el coste temporal.

Solución: Si consideramos que cada objeto tiene un valor igual a 1, el problema se reduce a la mochila discreta (maximizar el valor de los objetos considerando un valor unitario para cada uno es equivalente a maximizar el número de objetos) PERO considerando únicamente soluciones cuya suma de pesos sea exactamente igual a W.

$$X = \{(x_1, x_2, ..., x_N) \in \{0,1\}^N \mid \sum_{1 \le i \le N} x_i w_i = W\}$$

$$f((x_1, x_2, ..., x_N)) = \sum_{1 \le i \le N} x_i$$

$$\hat{x} = \operatorname*{argmax}_{x \in X} \sum_{1 \le i \le N} x_i$$

Un estado se representará con una secuencia incompleta $(x_1, x_2, ..., x_k)$, con $k < N$. Tendrá un coste espacial de $O(N)$. El estado inicial se puede representar como una tupla vacía (?). Para que un estado sea solución la tupla deberá tener una talla igual a N y además que se cumpla la restricción de que la suma de los pesos de los objetos considerados sea exactamente igual a W. También se podría considerar que un estado es solución en cuanto se detecte que el peso de los objetos considerados hasta el momento ya es igual a W aunque $k < N$, ya que se puede completar la tupla de forma trivial considerando $x_i = 0$ para todo i mayor que k y obtener así la única solución factible alcanzable a partir de ese estado.

La función de ramificación dará lugar a dos nuevos estados: el estado que añade la decisión de no considerar el siguiente objeto $k + 1$ y, si la suma del peso de los objetos considerados hasta el momento

más el del nuevo objeto w_{k+1} no sobrepasa la capacidad de la mochila, también el estado que resulta al considerar su inclusión:

$$branch(x_1, x_2, ..., x_k, ?) = \{(x_1, x_2, ..., x_k, 0, ?), (x_1, x_2, ..., x_k, 1, ?) \text{ si } \sum_{1 \leq i \leq k+1} x_i w_i \leq W\}$$

Para que la comprobación de la condición de no sobrepasar la capacidad W sea eficiente, habrá que mantener en el estado el peso acumulado hasta el momento o, equivalentemente, la capacidad libre en la mochila.

Se puede añadir una condición más a la ramificación:

$$branch(x_1, x_2, ..., x_k, ?) = \begin{cases} (x_1, x_2, ..., x_k, 0, ?), & \text{si } (\sum_{1 \leq i \leq k} x_i w_i + \sum_{k+2 \leq i \leq N} w_i) \geq W \\ (x_1, x_2, ..., x_k, 1, ?), & \text{si } \sum_{1 \leq i \leq k+1} x_i w_i \leq W \end{cases}$$

de esta forma se detecta antes que el estado es no prometedor y no se genera.

Para este problema será de gran utilidad ordenar los objetos de mayor a menor valor unitario en un preproceso. En el problema que nos ocupa, al considerarse que todos los objetos tienen valor 1 esto es equivalente a ordenar los objetos de menor a mayor peso (con un coste $O(N \log N)$). De esta forma se puede mejorar la función de ramificación: si están ordenados, en el momento que se detecta durante la ramificación del estado $(x_1, x_2, ..., x_k)$ con $k < N$ que el objeto que ocupa la posición $k+1$ no cabe en la mochila, esto es, que $\sum_{1 \leq i \leq k+1} x_i w_i > W$, ninguno de los objetos restantes cabrá, por lo que el estado $(x_1, x_2, ..., x_k)$ generará una única solución factible con el prefijo $(x_1, x_2, ..., x_k)$ y el resto de objetos no seleccionados, $x_i = 0$ para i entre $k+1$ y N, si y sólo si el peso acumulado coincide con W.

Una posible cota trivial sería utilizar la idea de que todos los objetos que quedan por considerar rellenarán completamente la mochila. Esta cota se puede calcular de forma incremental así:

- En el estado inicial: $cota\_superior(?) = N$
- En un estado intermedio de forma incremental:

$$cota\_superior(x_1, x_2, \ldots, x_k, 1, ?) = cota\_superior(x_1, x_2, \ldots, x_k, ?)$$

$$cota\_superior(x_1, x_2, \ldots, x_k, 0, ?) = cota\_superior(x_1, x_2, \ldots, x_k, ?) - 1$$

Se puede calcular con coste $O(1)$ pero es muy poco ajustada.

Una cota superior más ajustada sería asumir que el espacio que queda libre en la mochila se puede rellenar (completamente) resolviendo el problema de la mochila con fraccionamiento con los objetos que quedan por considerar. Una simplificación a esta cota sería ir metiendo los objetos (en el orden de menor a mayor peso) hasta que el peso acumulado sea mayor o igual que la capacidad de la mochila

$$cota\_superior(x_1, x_2, \ldots, x_k, ?) = \sum_{1 \leq i \leq k} x_i + MochilaEnteraConExceso(W - \sum_{1 \leq i \leq k} x_i w_i, \{w_{k+1}, \ldots, w_N\})$$

donde $MochilaEnteraConExceso(W - \sum_{1 \leq i \leq k} x_i w_i, \{w_{k+1}, \ldots, w_N\})$ va añadiendo objetos desde la posición $k+1$ hasta que el peso acumulado sea mayor o igual que la capacidad libre de la mochila.

Un ejemplo: $W = 10.0$ y se tienen $N = 10$ objetos de pesos $w = [1, 1.1, 1.2, 1.7, 2, 2.9, 3, 3.5, 3.8, 4]$ (asumimos que los hemos ordenado previamente de menor a mayor peso). Para el estado inicial (?) la cota optimista sería 7 de la parte desconocida que son los objetos a meter hasta que la suma sea mayor o igual a 10 (con un peso acumulado de $1 + 1.1 + 1.2 + 1.7 + 2 + 2.9 + 3 = 12.9$): $cota\_superior(?) = 7$. Para el estado hijo (1?) la cota será la misma que la del padre: $cota\_superior(1?) = cota\_superior(?) = 7$. Y para el estado donde no metemos el primer objeto (de peso 1), la suma acumulada sería: parte conocida (0 objetos) + parte desconocida (6 objetos): $cota\_superior(0?) = 0 + 6 = 6$, pues $1.1 + 1.2 + 1.7 + 2 + 2.9 + 3 = 11.9$. Si lo hacemos de forma incremental: $cota\_superior(0?) = cota\_superior(?) - 1$ si $12.9 - w_1 \geq W$, en este caso, $12.9 - 1 \geq 10$; en el caso de que no ocurra así deberemos considerar añadir un objeto más que seguro que cubre el espacio que ha dejado libre el objeto que hemos dejado de considerar. Por ejemplo: $W = 9.0$ y $w = [1, 1.1, 1.2, 1.7, 2, 2.9, 3, 3.5, 3.8, 4]$: $cota\_superior(?) = 6$ (pues 1+1.1+1.2+1.7+2+2.9=9.9) y el hijo:

cota_superior(0?) = 6 pues aunque no metamos el objeto primero $1.1 + 1.2 + 1.7 + 2 + 2.9 = 8.9$ hemos de añadir el objeto siguiente.

2 **3 puntos**

Aplicar la técnica de ramificación y poda con poda explícita al siguiente problema: sean N trabajadores y N tareas. Cualquier trabajador puede realizar cualquier tarea, con una duración en unidades de tiempo que puede variar dependiendo de la asignación trabajador-tarea. Se necesita realizar todas las tareas asignando exactamente una tarea a cada trabajador y un trabajador a cada tarea, de forma que el tiempo total de realizar todas las tareas se minimice. Muestra la ejecución sobre la siguiente instancia:

	Tarea1	Tarea2	Tarea3	Tarea4
p1	9	6	7	7
p2	6	4	3	7
p3	6	8	4	8
p4	7	5	9	6

Por ejemplo, la posición (1,2) de la matriz de costes indica que el trabajador p1 necesita 6 unidades de tiempo para ejecutar la tarea 2.

a) Explica brevemente cómo vas a representar un estado incompleto y un estado solución. Pon un ejemplo sobre la instancia.
b) Explica qué cota optimista vas a emplear.
c) La traza se debe mostrar como el *conjunto de estados activos* de cada iteración del algoritmo indicando la cota optimista de cada estado y marcando el estado que se selecciona para la siguiente iteración. La traza debe indicar también si se actualiza la variable mejor solución y la aplicación, cuando corresponda, de la poda explícita. En caso de inicializar la mejor solución inicial a una solución arbitraria, debes indicar el algoritmo que usas y su coste.

Solución: Las soluciones a este problema son las permutaciones de los N elementos. Una solución se puede representar como una tupla $(x_1, x_2, \ldots, x_N)$ donde $x_i = j$ indica que el trabajador i realiza la tarea j. Por ejemplo, (2,1,4,3). Un estado intermedio será una solución incompleta $(x_1, x_2, \ldots, x_k)$, con $k < N$, que representa todas las soluciones que tienen ese prefijo.; por ejemplo, (3,1,?).

Una posible cota sería elegir el coste menor para cada trabajador, independiente o no que la tarea ya esté asignada. Este cálculo se puede hacer en un preproceso con coste $O(N^2)$ y se obtiene un vector de mínimos: $[6, 3, 4, 5]$. El estado inicial tiene una cota que se obtiene sumando estos mínimos: $(?, 6+3+4+5 = 18)$ con un coste $O(N)$. El resto de cotas se puede obtener de forma incremental con coste constante. La solución inicial se inicializa con un valor pésimo: $x =$ None y $fx = +\infty$.

$A_0 = \{(?, 18)*\}$
$A_1 = \{(1?, 21), (2?, 18)*, (3?, 19), (4?, 19)\}$
$A_2 = \{(1?, 21), (3?, 19), (4?, 19), (21?, 21), (23?, 18)*, (24?, 22)\}$
$A_3 = \{(1?, 21), (3?, 19)*, (4?, 19), (21?, 21), (24?, 22), (231?, 20), (234?, 22)\}$
$A_4 = \{(1?, 21), (4?, 19)*, (21?, 21), (24?, 22), (231?, 20), (234?, 22), (31?, 22), (32?, 20), (34?, 23)\}$
$A_5 = \{(1?, 21), (21?, 21), (24?, 22), (231?, 20), (234?, 22), (31?, 22), (32?, 20), (34?, 23), (41?, 22), (42?, 20), (43?, 19)*\}$
$A_6 = \{(1?, 21), (21?, 21), (24?, 22), (231?, 20)*, (234?, 22), (31?, 22), (32?, 20), (34?, 23), (41?, 22), (42?, 20), (431?, 21), (432?, 23)\}$

Al ramificar el estado seleccionado se obtiene la primera solución, $(2, 3, 1, 4)$, con un valor de 21, que mejora la solución actual: $x = (2, 3, 1, 4)$ y $fx = 21$, por lo que se entra en el proceso de poda explícita y se elimina aquellos estados con una cota menor o igual a 21, quedando:

$A_7 = \{(32?, 20)*, (42?, 20)\}$

Al ramificar el estado seleccionado se obtiene dos nuevos estados: (321?) con un cota igual a 22, que no puede mejorar a la solución actual, por lo que no se inserta en A, y el estado (324?) con cota 24, que tampoco se inserta.

$A_8 = \{(42?, 20)*\}$

Al ramificarlo se obtienen dos nuevos estados: $(421?, 24)$ y $(423?, 22)$, ninguno de ellos puede mejorar la solución. El algoritmo termina porque no quedan estados por explorar y devuelve la solución óptima x.

Otra opción es inicializar la mejor solución con una solución voraz: se puede obtener el mínimo para cada trabajador, en orden creciente de trabajador, siempre que esa tarea no esté asignada previamente. Este algoritmo devuelve la solución $x = (2, 3, 1, 4)$ para esta instancia, con un tiempo total igual a $fx = 6 + 3 + 6 + 6 = 21$. De esta forma, la traza quedaría:

$A_0 = \{(?, 18)*\}$

$A_1 = \{(2?, 18)*, (3?, 19), (4?, 19)\}$. El estado $(1?, 21)$ no se guarda porque no puede mejorar x.

$A_2 = \{(3?, 19), (4?, 19), (23?, 18)*\}$. Los estados $(21?, 21)$ y $(24?, 22)$ no se guardan.

$A_3 = \{(3?, 19)*, (4?, 19), (231?, 20)\}$. El estado $(234?, 22)$ no se guarda.

$A_4 = \{(4?, 19)*, (231?, 20), (32?, 20)\}$. Los estados $(31?, 22)$ y $(34?, 23)$ no se guardan.

$A_5 = \{(231?, 20), (32?, 20), (42?, 20), (43?, 19)*\}$. El estado $(41?, 22)$ no se guarda.

$A_6 = \{(231?, 20)*, (32?, 20), (42?, 20)\}$. Los nuevos estados $(431?, 21), (432?, 23)$ no se guardan.

Al ramificar el estado seleccionado se obtiene la primera solución, $(2, 3, 1, 4)$, con un valor de 21, que no mejora la solución actual, por lo que se desecha.

$A_7 = \{(32?, 20)*, (42?, 20)\}$

Al ramificar el estado seleccionado se obtiene dos nuevos estados: (321?) con un cota igual a 22, que no puede mejorar a la solución actual, por lo que no se inserta en A, y el estado (324?) con cota 24, que tampoco se inserta.

$A_8 = \{(42?, 20)*\}$

Al ramificarlo se obtienen dos nuevos estados: $(421?, 24)$ y $(423?, 22)$, ninguno de ellos puede mejorar la solución. El algoritmo termina porque no quedan estados por explorar y devuelve la solución óptima x.

3 **3 puntos**

Queremos empaquetar N objetos de pesos $w_1, \ldots, w_N$ gastándonos el menor dinero en las cajas. La tienda de cajas nos puede proveer de un lote de cajas siempre que **todas sean del mismo tipo** y nos ofrece una lista con la capacidad y el precio de cada tipo de caja. Un ejemplo de lista sería `[(25,3.5),(50,6.8),(70,9.3)]` que indica que hay cajas de 25Kg de capacidad que valen 3.5 euros cada una, otras de 50Kg a 6.8 euros la unidad y, finalmente, cajas de 70Kg a 9.3 euros.

Se pide realizar una función Python que reciba la lista de objetos y los datos del proveedor de cajas. La función debe devolver una tupla con el número de cajas necesarias y el tipo de caja. Si no hay solución deberá devolver (-1,None), mientras que para 0 cajas no hace falta especificar el tipo de la caja, con lo que puede devolver (0,None). Sigue una estrategia voraz e indica el coste del algoritmo desarrollado y si resuelve o no este problema de manera óptima.

Ejemplo:

```
tipos = [(25,3.5),(50,6.8),(70,9.3)]
W = [10,30,20,40,40,30,25,25,40]
print(empaquetar(W,tipos))
```

Una posible solución a esta instancia sería el valor `(4,70)` que corresponde a gastarse 37.2 euros en 4 cajas de 70Kg de capacidad.

Solución: Una forma de abordar el problema pasa por resolver bin-packing para cada tipo de caja y seleccionar la mejor solución, que en este caso es utilizar el tipo de cajas que sirva para empaquetar los objetos cuyo coste total sea el menor (multiplicando el número de cajas resultado de bin-packing por su coste). Esta estrategia voraz no proporciona la solución óptima, pero sí una aproximación razonable. En el algoritmo, además, se filtran los tipos de cajas que no pueden dar lugar a una solución, con una sencilla comprobación: si el objeto de mayor peso no cabe en las cajas de ese tipo, se elimina el tipo. También se podría mejorar todo el proceso ordenando una única vez los objetos, antes de llamar a bin-packing.

```
def binpacking(pesos,capacidad):
  cajas = []
  for peso in sorted(pesos,reverse=True):
    for pos,cap in enumerate(cajas):
      if peso<=cap:
        cajas[pos] -= peso
```

```
        break
    else:
      cajas.append(capacidad-peso)
  return len(cajas)

def empaquetar(pesos,tipos):
  if len(pesos)==0:
    return (0,None)
  maxpeso = max(pesos)
  cajas_necesarias = [(binpacking(pesos,capacidad),capacidad,precio)
                      for (capacidad,precio) in tipos if maxpeso<=capacidad]
  pvp,capacidad,unidades = min(((cantidad*precio,capacidad,cantidad)
                                for (cantidad,capacidad,precio) in cajas_necesarias),
                               default=(0,None,-1))
  return (unidades,capacidad)
```