UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Algorítmica (11593)

Primer parcial

21/noviembre/2022

etsinf

Escuela Técnica Superior de Ingeniería Informática

NOMBRE: _____ **NÚM.:** _____

1. Programación Dinámica — 5 puntos

En una cadena de montaje se construyen aparatos formados por N piezas que se deben montar en un orden prefijado. La cadena de montaje tiene P operarios de modo que cada uno recibe lo del operario anterior. Sabiendo que $N > P$ y que cada operario ha de montar al menos 1 pieza, debemos indicar qué piezas debe montar cada operario para minimizar la suma de la *carga total* de los operarios, siendo $c(i, j, p)$ la *carga* del operario p ($1 \leq p \leq P$) al montar las piezas entre i y j inclusive, con $1 \leq i \leq j \leq N$. Para ello, se pide:

a) Especifica formalmente el conjunto de soluciones factibles X, la función objetivo a minimizar f, y la solución óptima buscada $\hat{x}$.

b) Escribe una ecuación recursiva que calcule el mínimo valor de la carga total que se puede conseguir. ¿Qué llamada inicial de la ecuación recursiva devuelve el valor pedido?

c) Diseña un algoritmo iterativo (en Python 3 o en seudocódigo) que devuelva el *valor de la mínima carga*.

d) Indica cuál es el coste temporal y espacial del algoritmo diseñado justificando brevemente tu respuesta.

e) ¿Crees que sería posible reducir el coste espacial? Si es así, indica cuál sería dicho coste y justifícalo.

Solución: Este problema es el agrupamiento de N puntos en la recta real (N piezas en la cadena de montaje) en P grupos (P operarios). Podemos representar una solución como una P-tupla donde se indica, para cada grupo $1 \leq p \leq P$, la última pieza montada por ese operario. La "carga total" de la agrupación de las N piezas en P grupos viene dada por la suma de las cargas de cada una de las agrupaciones. Así, por ejemplo, si $N = 8$ y $P = 3$:

- La agrupación $\{\{n_1\}, \{n_2, n_3, n_4, n_5\}, \{n_6, n_7, n_8\}\}$, expresada como $(1, 5, 8)$, tiene una carga total $c(1, 1, 1) + c(2, 5, 2) + c(6, 8, 3)$.
- Esta otra agrupación, $\{\{n_1, n_2\}, \{n_3, n_4\}, \{n_5, n_6, n_7, n_8\}\}$, expresada como $(2, 4, 8)$, tiene una carga total $c(1, 2, 1) + c(3, 4, 2) + c(5, 8, 3)$.

a) Formalizamos el problema así:

$$X = \{(g_1, g_2, \ldots, g_P) \in [1..N]^+ \mid g_P = N, g_i < g_{i+1}, 1 \leq i < P\}$$

$$f((g_1, g_2, \ldots, g_P)) = c(1, g_1, 1) + \sum_{i=2}^{P} c(g_{i-1} + 1, g_i, i)$$

$$\hat{x} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_P) = \operatorname*{argmin}_{x \in X} f(x)$$

b) Sea $F(p, n)$ la mínima carga que se puede obtener agrupando las n primeras piezas en p grupos (fíjate que hay que dejar al menos una pieza para cada operario anterior):

$$F(p, n) = \begin{cases} c(1, n, 1), & \text{si } p = 1 \\ \min\limits_{p-1 \leq i < n} (F(p-1, i) + c(i+1, n, p)), & \text{en otro caso} \end{cases}$$

La llamada $F(P, N)$ proporciona el valor de la mínima carga que se puede obtener con una asignación válida.

c) Un algoritmo iterativo a partir de la anterior ecuación recursiva:

```
def calcular_carga(P, N, c):
      # P es el numero de grupos (operarios) y N el de piezas
      # c es una funcion que se puede llamar con 3 argumentos c(i,n,p)

      F = {} #F indexada de 1..P y de 1..N
      for n in range(1, N+1): F[1,n] = c(1,n,1)
      for p in range(2, P+1):
            for n in range(p, N+1):
                  F[p,n] = min( F[p-1,i] + c(i+1,n,p) for i in range(p-1,n) )
      return F[P,N]
```

d) El coste espacial es $O(PN)$, determinado por el tamaño del almacén de resultados intermedios F. El coste temporal viene dado por los dos bucles de tamaños P y N y por la minimización interna que se hace sobre un conjunto de talla $O(N)$, por lo que el coste temporal total es $O(PN^2)$.

e) Es posible reducir el coste espacial utilizando dos vectores columna de tamaño $O(N)$ pues para resolver el problema $F[p, n]$ tan solo son necesarios los valores de la columna anterior $F[p-1, i]$, siendo prescindibles los previos.

2. Programación Dinámica — 2.5 puntos

El problema de la mochila discreta sin fraccionamiento puede resolverse de forma óptima con una estrategia de programación dinámica. Recordemos su planteamiento básico: dados N productos con pesos unitarios (en kilos) w_1, w_2, ..., w_N y valores respectivos v_1, v_2, ..., v_N y una mochila con capacidad para W kilos, desemos cargar ésta de modo que no se exceda la capacidad de carga de la mochila y el beneficio sea máximo.

En este caso, te pedimos:

a) Completa la siguiente solución del problema de la mochila discreta, basada en Programación Dinámica, de forma que devuelva una lista con los objetos que intervienen en la solución del problema, junto con el valor óptimo obtenido.

Por ejemplo, ante la entrada:

```
w=[1,3,5,2,7]          # lista de pesos de los objetos
v=[120,25,60,45,10]    # lista de valores de los objetos
W=10                   # peso maximo
```

la ejecución de la función:

```
l,f = iterative_knapsack_objects(W, v, w)
```

deberá devolver para la lista de objetos `l`, los valores:

```
[1, 0, 1, 1, 0]            # 0-el objeto no se pone, 1-el objeto se pone
```

La solución que implementes deberá estar basada en el almacenamiento de una matriz B de backpointers (punteros hacia atrás). Indica cómo modificar el siguiente código:

```
 1 def iterative_knapsack_objects(W, v, w):
 2   V = {}
 3   objects = []
 4   for c in range(W+1):
 5     V[0,c] = 0
 6   for i in range(1, len(v)+1):
 7     V[i,0] = 0
 8     for c in range(1, min(W,w[i-1])):
 9       V[i,c] = V[i-1,c]
10     for c in range(w[i-1], W+1):
11       V[i,c] = max(V[i-1,c], V[i-1,c-w[i-1]] + v[i-1])
12
13   return objects, V[len(v), W]
```

b) Haz una traza de tu algoritmo para la instancia presentada, mostrando el valor final de las matrices V y B.

c) Indica el coste espacial y temporal, justificando tus respuestas, tanto del algoritmo dado en el enunciado como del algoritmo que hayas diseñado.

Solución:

a) Habrá que declarar la matriz de backpointers, B, del mismo tamaño que el almacén de punteros e ir rellenándola sobre la marcha, para hacer un postproceso final sobre ella.

```
def iterative_knapsack_objects (W, v, w):
    V, B = {}, {}
    for c in xrange(W+1):
        V[0,c], B[0,c] = 0, None
    for i in range(1, len(v)+1):
        V[i,0], B[i,0] = 0, (i-1, 0)
        for c in range(1, min(W,w[i-1])):
            V[i,c], B[i,c] = V[i-1,c], (i-1, c)
        for c in range(w[i-1], W+1):
            if V[i-1,c] > V[i-1,c-w[i-1]] + v[i-1]:
                V[i,c], B[i,c] = V[i-1,c], (i-1, c)
            else:
                V[i,c], B[i,c] = V[i-1,c-w[i-1]]+v[i-1], (i-1, c-w[i-1])
    objects = []
    (i, c) = (len(v), W)
    while B[i,c] != None:
        if B[i,c] != (i-1, c): objects.append(1)
        else: objects.append(0)
        (i, c) = B[i,c]
    objects.reverse()
    return objects, V[len(v), W]
```

b) El contenido de las matrices será:

```
#   0    1    2    3    4    5          #      0       1      2      3      4      5
V=[[0,   0,   0,   0,   0,   0], # 0    B=[[None,  (0,0), (1,0), (2,0), (3,0), (4,0)], # 0
   [0, 120, 120, 120, 120, 120], # 1       [None,  (0,0), (1,1), (2,1), (3,1), (4,1)], # 1
   [0, 120, 120, 120, 120, 120], # 2       [None,  (0,1), (1,2), (2,2), (3,2), (4,2)], # 2
   [0, 120, 120, 120, 165, 165], # 3       [*None,*(0,2),*(1,3), (2,3), (3,1), (4,3)], # 3
   [0, 120, 145, 145, 165, 165], # 4       [None,  (0,3), (1,1), (2,4), (3,2), (4,4)], # 4
   [0, 120, 145, 145, 165, 165], # 5       [None,  (0,4), (1,2), (2,5), (3,3), (4,5)], # 5
   [0, 120, 145, 180, 190, 190], # 6       [None,  (0,5), (1,3), (2,1), (3,4), (4,6)], # 6
   [0, 120, 145, 180, 190, 225], # 7       [None,  (0,6), (1,4), (2,2), (3,5), (4,7)], # 7
   [0, 120, 145, 180, 225, 225], # 8       [None,  (0,7), (1,5),*(2,3), (3,6), (4,8)], # 8
   [0, 120, 145, 205, 225, 225], # 9       [None,  (0,8), (1,6), (2,4), (3,7), (4,9)], # 9
   [0, 120, 145, 205, 225,*225]] #10       [None,  (0,9), (1,7), (2,5),*(3,8),*(4,10)]]#10
```

y el algoritmo devolverá la lista de objetos `[1, 0, 1, 1, 0]` con un valor total de `V[5,10]=225`.

c) La complejidad espacial del algoritmo presentado es $\Theta(NW)$, debido al almacén de resultados intermedios, y la complejidad temporal también es $\Theta(NW)$, debido a los dos bucles del algortimo. La complejidad de la versión con recuperación de los objetos seleccionados es la misma, puesto que el almacén de backpointers es del mismo tamaño que el almacén de resultados intermedios y el bucle de postproceso no añade complejidad temporal.

3. Vuelta atrás **2.5 puntos**

Se debe preparar una cena de bienvenida en una cumbre internacional. El personal de protocolo sabe que solo puede invitar a mandatarios cuyos países no estén enemistados. Ha acudido a la cumbre la representación de P países, y protocolo debe seleccionar S representantes de forma que todos los S países tengan buenas relaciones ($S < P$). Protocolo dispone de la información necesaria para seleccionar a los S representantes: una matriz de booleanos R, de tamaño $P \times P$, representando una relación simétrica pero no necesariamente transitiva. La relación $R[i, j]$ indica si los representantes de los países i y j tienen buenas relaciones entre sí. Nos piden encontrar S representantes con buena relación entre sí. Por ejemplo, si $P = 5$ y $S = 3$, y la matriz de relaciones R es:

```
# 1 y 2 son incompatibles
# 3 y 4 son incompatibles
T,F = True,False
#     0  1  2  3  4
R = [[T, T, T, T, T], # 0
     [T, T, F, T, T], # 1
     [T, F, T, T, T], # 2
     [T, T, T, T, F], # 3
     [T, T, T, F, T]] # 4
```

`[0, 2, 3]` sería una posible solución (no necesariamente es la única).

Implementa un algoritmo que, utilizando la técnica de Búsqueda con Retroceso, devuelva la solución pedida, si es que existe. Para ello:

1. Explica brevemente la estrategia a seguir.
2. Describe cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuántos elementos van a contener, valor posible de esos elementos).
3. Escribe un algoritmo (en Python 3 o en seudocódigo) que implemente la estrategia anterior. Indicando cuál sería la llamada inicial que harías para resolverlo.

Solución: La estrategia consiste en ir construyendo una solución de forma incremental, esto es, una tupla de S candidatos con buena relación entre sí. Para ello, empezaremos con la tupla vacía, añadimos el primer candidato (en el orden dado, numerados de 1 a P); se añade el siguiente candidato si se cumplen las restricciones (si el país tiene buenas relaciones con el primero); se añade el tercer candidato, si es compatible a los dos anteriores, y así sucesivamente. En el momento que el último candidato no sea compatible a cualquiera de los anteriores, se desestima, y se pasa a considerar el siguiente.

Una solución se puede representar, entre muchas otras posibilidades, como un vector de enteros con los representantes de los países seleccionados. Un estado incompleto será un vector con menos de S candidatos considerados para su inclusión.

Fíjate que, en el fondo, este problema no es más que la obtención de una combinación de P elementos tomados de S en S, a lo que se añade la restricción de que los elementos (los *países*) sean compatibles entre sí.

```
P = 5
S = 3
# 1 y 2 son incompatibles
# 3 y 4 son incompatibles
T,F = True,False
#     0  1  2  3  4
R = [[T, T, T, T, T], # 0
     [T, T, F, T, T], # 1
     [T, F, T, T, T], # 2
     [T, T, T, T, F], # 3
     [T, T, T, F, T]] # 4
```

```
for seleccion in protocolo(P, S, R):
    print(seleccion)

def protocolo(P, S, R):
    # P es el nº de países
    # S los que hay que seleccionar, S<P
    # R es una matriz PxP
    solucion = [None]*S
    def backtracking( seleccionados ):
        if seleccionados >= S:
            return solucion
        desde = 0 if seleccionados==0 else solucion[seleccionados-1]+1
        for pais in range(desde,P):
            if all(R[solucion[i],pais] for i in range(seleccionados)):
                solucion[seleccionados] = pais
                resul = backtracking( seleccionados+1 )
                if resul is not None:
                    return resul
    # llamada inicial
    return backtracking(0)
```

Da estas 3 soluciones, aunque se pide que dé una sola:

```
[0, 1, 3]
[0, 1, 4]
[0, 2, 3]
[0, 2, 4]
```