Prueba de aula Algorítmica (11593)

15 de noviembre de 2022

1 **2 puntos**

El siguiente algoritmo iterativo de Programación Dinámica resuelve el problema del cálculo del precio del trayecto más barato en el río Congo:

```
def iterative_minimum_price(E, c):
   C = [None]*E
   C[0] = 1
   C[1] = c(0,1)
   for i in xrange(2, E):
      C[i] = min( C[i-1] + c(i-1,i), C[i-2] + c(i-2,i) )
   return C[E-1]
```

siendo `E` el número de embarcaderos y `c(i,j)` una función que devuelve el coste de un embarcadero `i` a otro `j` (ambos numerados de `0` a `E-1`). Responde a las siguientes cuestiones:

a) Analiza el coste temporal y espacial del algoritmo.
b) Modifica el código para conocer también el camino de mínimo coste.

Solución: El coste espacial viene dado por el vector `C`, de talla `E`: $O(E)$, y el temporal por el bucle `for` que se ejecuta $O(E)$ veces, cada una de ellas con un coste constante (es una minimización entre dos elementos, independiente de la talla del prroblema). Por tanto, también $O(E)$.

Para encontrar el camino de mínimo precio basta con añadir un vector de punteros hacia atrás y luego hacer sobre él el correspondiente postproceso para recuperar el camino:

```
def iterative_minimum_price(E, c):
   C,B = [None]*E,[None]*E
   C[0] = 1
   C[1] = c(0,1)
   for i in xrange(2, E):
      C[i],B[i] = min( (C[i-1] + c(i-1,i),i-1), (C[i-2] + c(i-2,i),i-2) )
   path = [E-1]
   while B[path[-1]] != None:
     path.append(B[path[-1]])
   path.reverse()
   return C[E-1],path
```

Deseamos asfaltar los K kilómetros de carretera que separan dos ciudades. Tenemos N tipos de tramos con los que construir la carretera. Para cada tipo de tramo i ($1 \leq i \leq N$) nos indican:

- su longitud en kilómetros L_i,
- su coste en euros C_i,
- el número de tramos en stock S_i.

Los tramos deben ser utilizados enteros y nos hemos de limitar al stock. Queremos conseguir tramos que sumen exactamente los K kilómetros minimizando el coste total. **Se pide:**

1. Especificar formalmente el conjunto de soluciones factibles X, la función objetivo a minimizar f y la solución óptima buscada $\hat{x}$.
2. Una ecuación recursiva que resuelva el problema y la llamada inicial.
3. El algoritmo iterativo asociado a la ecuación recursiva anterior para calcular *el menor coste* (no hace falta determinar el número de tramos necesarios de cada tipo).
4. El coste temporal y espacial del algoritmo iterativo.

Solución:

Podemos representar las soluciones mediante una tupla de longitud N donde cada elemento indica el número de tramos de cada tipo:

$$X = \left\{ (x_1, \ldots, x_N) \in \mathbf{N}^N | 0 \leq x_t \leq S_t, 1 \leq t \leq N, \sum_{t=1}^{N} x_t L_t = K \right\}$$

La función objetivo es $f((x_1, \ldots, x_N)) = \sum_{t=1}^{N} x_t C_t$, mientras que la solución óptima buscada es $\hat{x} = \operatorname{argmin}_{x \in X} f(x)$.

Una ecuación recursiva que resuelva el problema:

$$M(t,k) = \begin{cases} 0, & \text{si } t = 0 \text{ y } k = 0 \\ \infty, & \text{si } t = 0 \text{ y } k > 0 \\ \min\limits_{0 \leq x_t \leq \min(S_t, \lfloor k/L_t \rfloor)} M(t-1, k - x_t L_t) + x_t C_t, & \text{en otro caso} \end{cases}$$

La llamada inicial sería $M(N, K)$. Este problema es virtualmente idéntico al cambio de monedas con limitación del número de monedas, excepto que el coste de usar cada moneda (aquí un tramo) varía según su tipo.

```
def tramos(K,L,C,S):
  # K es un entero
  # L, C, S son listas de longitud N
  N, inf = len(L), 2**31
  M = { (0,k):inf for k in range(K+1) }
  M[0,0] = 0 # caso base
  for t in range(N): # tipo de tramo
    for k in range(K+1): # distancia
      M[t,k] = min(M[t-1,k-x*L[i]]+x*C[i] for x in range( min(k//L[i],S[i]) +1))
  return M[N,K]
```

El coste espacial de este algoritmo es $O(NK)$, mientras que el coste temporal es $O(NK^2)$ que corresponden a los costes vistos en teoría para el problema del cambio de monedas con limitación del número de monedas puesto que hacer distinto el coste de cada tramo no varía la complejidad espacial o temporal del algoritmo.

3 **3 puntos**

En el problema de la existencia de un ciclo Hamiltoniano, dado un grafo $G = (V, E)$, deseamos encontrar un camino que parta de un vértice, termine en el mismo vértice y visite todos los vértices del grafo una sola vez (excepto en el caso del vértice de partida, que deberá coincidir con el de llegada).

Se pide que implementes un algoritmo que, utilizando la técnica de Búsqueda con Retroceso, devuelva la solución encontrada, si es que existe. Utiliza una representación de grafo en forma de listas de adyacencia. Recuerda que, como el ciclo hamiltoniano puede empezar en cualquier vértice, es una práctica habitual empezar la comprobación por el primer vértice del grafo, $G.V[0]$. Para ello:

1. Explica brevemente la estrategia a seguir.
2. Escribe un algoritmo (en seudo-código o python) que implemente la estrategia anterior.

Solución:

```
# un grafo dirigido representado mediante listas de adyacencia seria
# simplemente una lista de listas del tipo:

G = [[1,2,3],       # del vertice 0 vamos a los vertices 1,2,3
     [0,3,4],       # del vertice 1
     [0,3,5],       # del vertice 2
     [0,1,2,4,5,6], # del vertice 3
     [1,3,7],       # del vertice 4
     [2,3,6,8],     # del vertice 5
     [3,5,7,8,9],   # del vertice 6
     [4,6,9],       # del vertice 7
     [5,6,9],       # del vertice 8
     [6,7,8]]       # del vertice 9

# para este tipo de grafo se puede definir el siguiente codigo de backtracking:

def hamiltonian_cycle(G):
  def backtracking(path):
    if len(path)==len(G):
      if path[0] in G[path[-1]]: return path+[0]
    else:
      for v in [x for x in G[path[-1]] if x not in path]:
        found = backtracking(path+[v])
        if found!=None: return found
    return None
  return backtracking([0])

# ejemplo de uso:
print hamiltonian_cycle(G)
```