UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Algorítmica (11593)

Primer Parcial
8 de noviembre de 2018

etsinf
Escuela Técnica Superior de Ingeniería Informática

NOMBRE:

1 — 5 puntos

Dados N puntos distintos de la recta real $x_1, \ldots, x_N$ ya ordenados: $x_1 < x_2 < \cdots < x_N$, queremos agruparlos en conjuntos de **puntos consecutivos** con al menos un punto por grupo: $C_1, C_2, \ldots, C_m$. Al haber al menos un punto por grupo, es obvio que $m \leq N$. Por ejemplo, para 8 puntos $(x_1, x_2, \ldots, x_8)$, dos posibles agrupaciones serían:

a) $C_1 = \{x_1, x_2, x_3\}, C_2 = \{x_4\}, C_3 = \{x_5, x_6\}, C_4 = \{x_7, x_8\}$

b) $C_1 = \{x_1, x_2\}, C_2 = \{x_3, x_4, x_5, x_6, x_7\}, C_3 = \{x_8\}$

De todas las formas posibles de realizar esta agrupación, queremos una que maximice la siguiente función de bondad del agrupamiento:

$$\sum_{k=1}^{m} \alpha(s_k, e_k)$$

donde $\alpha(a, b)$ con $1 \leq a \leq b \leq N$ mide lo bueno que es agrupar los puntos $x_a, x_{a+1}, \ldots, x_b$ en un mismo grupo, y donde s_k y e_k son los extremos (s de *start*, e de *end*) del grupo C_k. Por ejemplo, la bondad calculada para los ejemplos anteriores sería, respectivamente:

a) $\alpha(1,3) + \alpha(4,4) + \alpha(5,6) + \alpha(7,8)$

b) $\alpha(1,2) + \alpha(3,7) + \alpha(8,8)$

Se pide:

1. Especificar formalmente el conjunto de soluciones factibles X, la función objetivo a maximizar f y la solución óptima buscada $\hat{x}$.
2. Una ecuación recursiva que calcule el máximo valor de bondad de la mejor agrupación.
3. El algoritmo iterativo (preferiblemente en Python3) asociado a la ecuación recursiva anterior para calcular el máximo valor de bondad.

Solución:

1. Cada solución factible se puede representar mediante una lista de índices asociados al primer punto de cada grupo:

   $$X = \{(s_1, \ldots, s_m) \in [1..N]^+ \mid s_j < s_{j+1}, \forall 1 \leq j < m\}$$

   La función objetivo a maximizar es $f((s_1, \ldots, s_m)) = \left(\sum_{k=1}^{m-1} \alpha(s_k, s_{k+1} - 1)\right) + \alpha(s_m, N)$

   La solución óptima buscada es $\hat{x} = \text{argmax}_{x \in X} f(x)$.

2. Vamos a considerar que $F(k)$ calcula el mejor beneficio que se puede conseguir creando un número cualquiera de grupos con la restricción de que el último de esos grupos termina en x_k salvo que k sea 0 en cuyo caso significa que no se contempla ningún punto:

   $$F(k) = \begin{cases} 0, & \text{si } k = 0 \\ \text{máx}_{0 \leq j < k} F(j) + \alpha(j+1, k), & \text{en otro caso} \end{cases}$$

   La llamada inicial sería $F(N)$.

3. El algoritmo iterativo para calcular el máximo valor de bondad puede realizarse utilizando un diccinario para almacenar los resultados intermedios:

```
def grupos(N,alpha):
    F = { 0:0 }
    for k in range(1,N+1):
        F[k] = max(F[j]+alpha(j+1,k) for j in range(k))
    return F[N]
```

2 **2.5 puntos**

Debemos realizar A actividades y, para cada una, hay que elegir una modalidad entre 1 y M. Actividades diferentes pueden elegir una misma modalidad.

La función points(a, m) nos da la puntuación que genera la actividad a en la modalidad m (con $1 \leq a \leq A$ y con $1 \leq m \leq M$), mientras que dur(a, m) indica la duración correspondiente.

El siguiente código calcula la máxima puntuación alcanzable teniendo en cuenta la restricción adicional de que la suma total de las duraciones ha de ser exactamente D:

```
def elegir(A,M,D,points,dur,infty=2**31):
    P = {}
    # P[a,d] es la máxima puntuación asociada a las a primeras
    # actividades habiendo usado en ellas una duración total d
    P[0,0] = 0
    for d in range(1,D+1):
        P[0,d] = -infty
    for a in range(1,A+1):
        for d in range(D+1):
            P[a,d]=max((P[a-1,d-dur(a,m)]+points(a,m)
                        for m in range(1,M+1) if d>=dur(a,m)),
                       default=-infty)
    return P[A,D]
```

Se pide:

1. Realizar los cambios necesarios que consideres para que la función devuelva adicionalmente a la puntuación máxima, el conjunto de modalidades asociado a cada actividad.

2. Calcular el coste temporal y espacial del algoritmo iterativo proporcionado, así como el diseñado por tí, justificando (brevemente) las respuestas.

Solución:

1. Cambios necesarios que devolver la modalidad asociada a cada actividad:

```
def elegir(A,M,D,points,dur,infty=2**31):
    P,B = {},{}
    # P[a,d] es la máxima puntuación asociada a las a primeras
    # actividades habiendo usado en ellas una duración total d
    P[0,0],B[0,0] = 0,None
    for d in range(1,D+1):
        P[0,d],B[0,d] = -infty,None
    for a in range(1,A+1):
        for d in range(D+1):
            P[a,d],B[a,d]=max(((P[a-1,d-dur(a,m)]+points(a,m),m)
                               for m in range(1,M+1) if d>=dur(a,m)),
                              default=(-infty,None))
    a,d,resul = A,D,{}
    while B[a,d] != None:
        resul[a] = B[a,d]
        d -= dur(a,B[a,d])
        a -= 1
    return P[A,D],resul
```

2. El coste temporal del algoritmo proporcionado en el enunciado es claramente $O(A \cdot D \cdot M)$ ya que se rellena una matriz de resultados intermedios de talla $A \times D$ y para cada elemento hay que iterar en un bucle `for m in range(1,M+1)`, siendo constante el coste de cada una de esas iteraciones. El coste espacial del algoritmo proporcionado es $O(A \cdot D)$, debida a la matriz de resultados intermedios.

   El coste temporal es *pseudo-polinómico* puesto que aunque tanto la talla de points como dur dependen de $A \times M$, el valor D no refleja la talla del problema al ser un valor concreto (cuya talla es realmente $\log_2 D$).

   Por otra parte, tanto el coste temporal como el espacial del algoritmo, tras añadir la parte de recuperar la modalidad asociada a cada actividad, para la secuencia de mayor puntuación, sigue siendo $O(A \cdot D \cdot M)$ y $O(A \cdot D)$, respectivamente, puesto que la parte que recupera el camino tiene un coste espacial $O(1)$ y un coste temporal $O(A)$. Estos costes adicionales están dominados por el coste de la etapa anterior.

3 — 2.5 puntos

Todos conocemos el problema de las N-reinas: se trata de situar N reinas en un tablero de ajedrez de $N \times N$ casillas o escaques, de manera que ninguna reina amenace a las demás. El espacio de búsqueda o conjunto de soluciones factibles se puede formalizar de la manera siguiente:

$$X = \{(r_0, r_1, \ldots, r_{N-1}) \in [0..N-1]^N \mid r_k \neq r_l \wedge |r_l - r_k| \neq l - k, 0 \leq k < l < N\}$$

donde r_i indica la fila (*row*) en que se sitúa la reina de la columna i-ésima. El siguiente código encuentra una solución para este problema:

```
def nqueens(N):
    def is_promising(s, newrow):
        # determine if a new queen can be put in newcol,newrow
        newcol = len(s)
        return all(newrow != row and newcol-col != abs(newrow-row)
                   for (col,row) in enumerate(s))
    def backtracking(s):
        if len(s) == N: return s
        for row in range(N):
            if is_promising(s, row):
                found = backtracking(s+[row])
                if found != None: return found
        return None
    return backtracking([])
```

Nos han propocionado un tablero donde cada una de las $N \times N$ casillas tiene asociado un color (no necesariamente diferente). Para indicar el color de la casilla nos dan un argumento adicional llamado `colormap` de modo que `colormap[row][col]` indica el color de esa coordenada del tablero. En el nuevo tablero, ahora nos exigen añadir al problema una restricción adicional: no solamente las reinas no deben amenazarse entre sí sino que, además, *cada reina debe situarse encima de una casilla de color diferente a las demás reinas.*

Se pide:

1. Modificar la formalización del espacio de búsqueda X para que tenga en cuenta la nueva restricción.

2. Adaptar el código anterior para que también tenga en cuenta la nueva restricción. La cabecera de la función será, por tanto:

   ```
   def nqueens_color(N,colormap):
       ...
   ```

Así, por ejemplo, la llamada a `nqueens(4)` daría como resultado `[1, 3, 0, 2]`, mientras que con el siguiente `colormap` (donde `colormap[row][col]` devolvería una letra en este ejemplo):

```
colormap = ["rgxy",
            "rgxy",
            "xgyr",
            "gyrx"]
```

la llamada a `nqueens_color(4,colormap)` dará un resultado distinto: `[2, 0, 3, 1]`.

Solución:

1. Modificar la formalización del espacio de búsqueda:

$$X = \{(r_0, r_1, \ldots, r_{N-1}) \in [0..N-1]^N \mid r_k \neq r_l \wedge |r_l - r_k| \neq l - k, \\ \text{colormap}[r_k][k] \neq \text{colormap}[r_l][l]\ 0 \leq k < l < N\}$$

2. Adaptar el código para tener en cuenta la nueva restricción:

```
def nqueens_color(N,colormap):
    def is_promising(s, newrow):
        # check is a new queen can be put in newcol,newrow
        newcol = len(s)
        return all(newrow != row and newcol-col != abs(newrow-row) and
                   colormap[row][col] != colormap[newrow][newcol]
                   for (col,row) in enumerate(s))
    def backtracking(s):
        if len(s) == N: return s
        for row in range(N):
            if is_promising(s, row):
                found = backtracking(s+[row])
                if found != None: return found
        return None
    return backtracking([])
```