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.

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.

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]`.