MaxiSubconjunto

Enunciado

Dada una matriz simétrica M de $n \times n$ números naturales, y un número k, queremos encontrar un subconjunto I de $\{1, ..., n\}$ con $|I| = k$ que maximice $\sum_{i,j \in I} M_{i,j}$. Por ejemplo, si $k = 3$ y

$$M = \begin{pmatrix} 0 & 10 & 10 & 1 \\ - & 0 & 5 & 2 \\ - & - & 0 & 1 \\ - & - & - & 0 \end{pmatrix},$$

entonces $I = \{1, 2, 3\}$ es una solución óptima.

a) Diseñar un algoritmo de *backtracking* para resolver el problema, indicando claramente cómo se codifica una solución candidata, cuáles soluciones son válidas y qué valor tienen, qué es una solución parcial y cómo se extiende cada solución parcial.

b) Calcular la complejidad temporal y espacial del mismo.

c) Proponer una poda por optimalidad y mostrar que es correcta.

Resolución

a) Usando índices entre 0 y $n - 1$ como hace Python, en vez de 1 al n como dice el enunciado, un algoritmo de backtracking que soluciona el problema es:

```python
def f(M: list[list[int]], k: int) -> list[int]:
  n = len(M)
  def value(I: list[int]) -> int:
    return sum(M[i][j] for i in I for j in I)

  def g(I: list[int], i: int) -> list[int] | None:
    if len(I) == k: return I
    if i == n: return None
    x = g(I, i + 1)
    y = g(I + [i], i + 1)
    if y is None: return x
    if x is None: return y
    if value(x) >= value(y): return x
    return y

  return g([], 0)
```

La semántica de `g` es `g(I, i)` es la mejor forma de completar `I`, usando sólo números mayores o iguales a `i`. Acá `I` es una sub-solución, es decir, una lista de a lo sumo `k` índices de `M`. Por ejemplo, si `k = 3`, `I = [1]` es una sub-solución, como también lo es `I = [0, 1, 2]`. Extender una sub-solución a otra sub-solución es agregar números a la lista. Una sub-solución `I` es solución cuando `len(I) == k`, y ahí tiene valor `value(I)`.

b) Podemos acotar la complejidad del algoritmo tomando como peor caso $k = n$. Esto es porque mientras más bajo k, más temprano llegamos a `len(I) == k` y terminamos. Luego, haciendo k lo más grande posible (n) hacemos que nuestro algoritmo tarde lo máximo posible. De hecho, como `len(I) <= i`, y `i <= n`, al tener `len(I) == k == n`, debemos tener `i == n`, y tenemos una sola guarda: `if i == n: return None`.

Sabiendo eso, el comportamiento de nuestro código es relativamente simple: Cada vez hacemos dos llamadas, con `i + 1`, y cuando `i == n`, el árbol termina. Luego tenemos exactamente 2^n

vértices en nuestro árbol de recursión, y al hacer $O(n)$ trabajo en cada vértice (evaluando `value` dos veces), la complejidad temporal asintótica en el peor caso es $O(n2^n)$.

Como máximo vamos a tener n niveles de stack, y en cada nivel de stack vamos a tener una lista (`I` o `I + [i]`), de tamaño a lo sumo n. Luego usamos $O(n^2)$ espacio.

c) Cada vez que decidimos no usar un elemento, acotamos por arriba el máximo valor de cualquier extensión de nuestra sub-solución actual. El invariante que vamos a mantener es que `cota == value(I + list(range(i, n)))`. Es decir, `cota` es el valor que obtendríamos si, empezando con esta sub-solución actual `I`, agregamos *todos* los índices mayores o iguales a `i`.

Claramente podemos tomar `cota = value(list(range(n)))` como cota inicial, porque es cota superior de todas las soluciones. En particular, es la cota correcta para mantener el invariante que dijimos arriba, dado que empezamos con `I = []`.

Cuando decidimos no-usar un elemento, tenemos que restarle a `cota` todos los elementos que nunca van a formar parte de una extensión a `I`. Estos van a ser todos los elementos que podríamos haber tenido en nuestra suma final hasta ahora, pero a partir de ahora, sabemos que nunca van a estar. Si el elemento que decidimos no-usar es i, los elementos que no vamos a usar son $M_{i,j}$ y $M_{j,i}$ tales que $j \geq i$, o $j \in I$. Veámoslo con un ejemplo:

$$M = \begin{pmatrix} 10 & 4 & 10 & 2 & 9 \\ 4 & 10 & 2 & 2 & 6 \\ 10 & 2 & 9 & 9 & 6 \\ 2 & 2 & 9 & 6 & 4 \\ 9 & 6 & 6 & 4 & 1 \end{pmatrix}$$

Si tenemos `I = [0]`, `i = 2`, entonces ya decidimos no usar la segunda fila ni la segunda columna (es decir, el índice 1), y luego `cota` ya tiene restados los números en rojo:

$$\begin{pmatrix} 10 & 4 & 10 & 2 & 9 \\ 4 & 10 & 2 & 2 & 6 \\ 10 & 2 & 9 & 9 & 6 \\ 2 & 2 & 9 & 6 & 4 \\ 9 & 6 & 6 & 4 & 1 \end{pmatrix}$$

Tenemos `cota = 10 + 10 + 2 + 9 + 10 + 9 + 9 + 6 + 2 + 9 + 6 + 4 + 9 + 6 + 4 + 1 = 106`. Al decidir no usar el índice `i = 2`, encontramos que no vamos a poder usar lo que marcamos ahora con azul:

$$\begin{pmatrix} 10 & 4 & 10 & 2 & 9 \\ 4 & 10 & 2 & 2 & 6 \\ 10 & 2 & 9 & 9 & 6 \\ 2 & 2 & 9 & 6 & 4 \\ 9 & 6 & 6 & 4 & 1 \end{pmatrix}$$

Cuánto decrece la cota? La suma de

1) Las celdas de la fila 2, cuya columna sea más grande que `i` (porque la cota sólo suma cosas que vengan después de o exactamente en `i`, recordar que `cota == value(I + list(range(i, n)))`), y que no haya ya cruzado con rojo.
2) Análogamente, las celdas de la columna 2, cuya fila sea más grande que `i`.
3) Las celdas que tengan una fila igual a `i`, y columna en `I`.
4) Análogamente, las celdas que tengan columna igual a `i`, y fila en `I`.

Esto es restarle `10 + 9 + 9 + 6 + 10 + 9 + 6 = 2 * (10 + 9 + 9 + 6) - 9 = 2 * sum(M[i][j] for j in range(n) if j >= i or j in I) - M[i][i]`. Las celdas azules de la primer fila y la primer columna (los dos "10" azules) las restamos por `j in I`, las otras por `j >= i`.

También vamos a aprovechar y computar el valor de la sub-solución actual a medida que hacemos recursión.

```
def fc(M: list[list[int]], k: int) -> list[int]:
  n = len(M)

  def value(I: list[int]) -> int:
    return sum(M[i][j] for i in I for j in I)

  S = value(list(range(n)))
  P = [sum(M[i]) for i in range(n)]

  mejor_solucion_hasta_ahora = []
  mejor_valor_hasta_ahora = 0

  def g(I: list[list[int]], i: int,
        valI: int, cota: int) -> list[int] | None:
    nonlocal mejor_valor_hasta_ahora, mejor_solucion_hasta_ahora
    if len(I) == k:
      if valI > mejor_valor_hasta_ahora:
        mejor_valor_hasta_ahora = valI
        mejor_solucion_hasta_ahora = I
      return
    if i == n or cota < mejor_valor_hasta_ahora:
      return

    # Rama 1: Decidimos no-agregar i a I.
    nueva_cota = cota - 2 * sum(M[i][j] for j in range(n)
                                if j >= i or j in I) + M[i][i]
    g(I, i + 1, valI, nueva_cota)
    # Rama 2: Decidimos agregar i a I.
    nuevos_valores = M[i][i] + 2 * sum(M[k][i] for k in I)
    g(I + [i], i + 1, valI + nuevos_valores, cota)

  g([], 0, 0, S)
  return mejor_solucion_hasta_ahora
```

Los invariantes que mantenemos son que `valI = value(I)`, `len(I) <= i <= n`, que todos los elementos de `I` son menores estrictos que `i`, que `cota == value(I + list(range(i, n)))`, y que `valI <= cota`.

Podemos ver que esta cota es efectiva, cuando frecuentemente tendríamos que explorar el árbol entero de otra forma (es decir, cuando k es grande):

```
import random
n = 15
M = [[0]*n for _ in range(n)]

for i in range(n):
  for j in range(i + 1):
    M[i][j] = random.randint(1, 10)
    M[j][i] = M[i][j]
```

```
%timeit f(M, 10)
# 49.7 ms ± 1.26 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
%timeit fc(M, 10)
# 15.2 ms ± 3.72 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Sin embargo, cuando k es chico, el costo adicional de calcular la cota ralenta más el programa que lo que podemos acelerar podando ramas:

```
%timeit f(M, 2)
# 210 µs ± 37.7 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
%timeit fc(M, 2)
# 265 µs ± 6.21 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Esto nos recuerda que toda poda tiene que ser evaluada en la práctica, y no por recortar vértices necesariamente estamos haciendo nuestro programa más rápido.