

MaxiSubconjunto

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

$$M = \begin{pmatrix} 0 & 10 & 10 & 1 \\ - & 0 & 5 & 2 \\ - & - & 0 & 1 \\ - & - & - & 0 \end{pmatrix}, \quad \begin{array}{l} \{1,2,3\} \{1,3,4\} \{1,2,4\} \\ \{2,3,4\} \{2,1,4\} \end{array}$$

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.

① Cuáles son las soluciones candidatas? Todos los subconjuntos de $\{1, \dots, n\}$ de k elementos

Cuáles son los hijos? Aquellos que maximizan $\sum_{i,j \in I} M_{ij}$ y tengan elementos distintos

Cuáles son las soluciones y cómo se extiende?

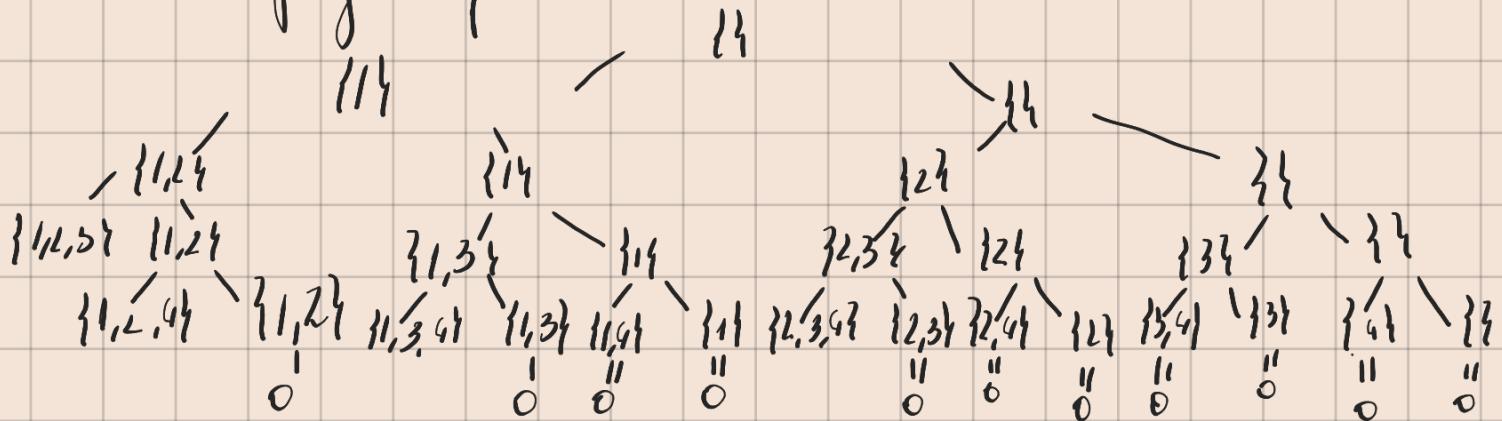
Son rectas de tamaño $i+1$

Primero fijo v_1 , luego elijo $v_2 / v_2 \neq v_1$, luego $v_3 / v_3 \neq v_2 \neq v_1$ y así
y me fijo I y maximizo $\sum_{i,j \in I} M_{ij}$

Case ej: fijo $v_1 \rightarrow \langle 1, \quad , \quad \rangle$
 elijo $2, 3 \rightarrow \langle 1, 2, \quad \rangle$ elijo de acuerdo
 abajo arriba

$$v_{1,1} = n \text{ qd, } v_{1,2} = n-1 \text{ qd}$$

Supongamos que de I a n está ordenado



MS(M, k, i, conj) = $\begin{cases} \text{conj} \\ \emptyset \\ \max(\text{SumaConjMatriz}(M, MS(M, k-1, i+1, \text{conj} \oplus \{i\})) \\ \text{SumaConjMatriz}(M, MS(M, k-1, i+1, \text{conj}))) \end{cases}$
 → hace la suma de las pos de la matriz

MS(M, k, i, conj)

if $k = 0$ then

return conj

endif

if $k >$ tamaño fila $M - i$

return conj vacío

else

return max(SumaConjMatriz($M, MS(M, k-1, i+1, \text{conj} \cup \{i\})$, SumaConjMatriz($M, MS(M, k, i+1, \text{conj})$)).pi2
 ↗ compara el 1^{er} elemento de la tupla
 y devuelve la que tenga mas en ese
 y despues devuelve v → podríamos
 hacer mejor modificando v ^{referencia}

endif

SumaConjMatriz(matrix m, vector &v)

Suma = 0

For i in v.size()

Suma += m[v[i]][v[i]]

For j = i+1 in v.size()

Suma += m[v[i]][v[j]] * 2

Endfor

Endfor

Return (suma, v)

Estamos aprovechando la matriz simétrica

El algoritmo anterior es erróneo porque está mal planteado, no queremos elegir el número máximo, sino el conjunto que maximiza esa suma

Reinvocamos la función que habíamos armado

SumaConjMatriz(matrix m, vector &v)

 Suma = 0

 For i in v.size()

 Suma += m[v[i]][v[i]]

 For j = i+1 in v.size()

 Suma += m[v[i]][v[j]]*2

 Endfor

Endfor

Return suma

Estamos aprovechando la matriz simétrica

Ahora planteamos el algoritmo

¿Cuáles van a ser mis casos base?

Si encontramos un conjunto de k elementos que maximice mi suma

Si $k = 0$ quiero ver si maximiza mi suma porque ya habría encontrado un conjunto

Si la cantidad de elementos que me quedan no completan k no sigo buscando porque no me lleva a ningún lado

```

vector mejorSol = {}
MS(&M, k, i, &conj)
if k = 0 then
    if sumaConjMatriz(&M, conj) mayor que
        sumaConjMatriz(&M, mejorSol) then
            mejorSol = conj
}
else
    if k menor (M.tamaño_fila()-i)
        conj.push_back(i)
        MS(M,k-1,i+1,conj)
        conj.pop_back(i)
        MS(M,k-1,i+1,conj)
}

```

Este sería un código correcto, ahora vamos a explicar por qué

1. Siempre que queremos el conjunto que implique tal cosa lo mejor que podemos hacer es tener una variable global donde lo vayamos almacenando y cambiando según corresponda
 2. Si encontramos un conjunto de k elementos nos tenemos que fijar si es mejor o peor que el que ya teníamos, es decir, si maximiza o no la función
 3. Si tenemos elementos para poner en el conjunto seguimos iterando, sino, cortamos las recursiones.
- Supongamos que quedan 4 números pero k = 10, seguir por este camino no lleva a soluciones factibles porque nos va a quedar un $k = \{\text{elementos anteriores}\} \cup \{\text{estos nuevos 4 elementos}\}$ pero necesitaríamos agregar 6 más para una posible solución.

3.1. ¿Por qué hacemos ese push y ese pop?

Siguiendo el ejemplo que hicimos arriba, el backtracking debe explorar de forma sistemática todas las soluciones posibles. En nuestra recursión primero va a explorar hasta llegar a la hoja {1,2,3}. Una vez que termine con ese caso, el backtracking va a decir: "ok, esta es una respuesta posible pero si cambiamos el 3 por otro valor va a aparecer una nueva, explórala". Entonces vamos a tener que volver al nodo del cuál vinimos y explorar la nueva solución. La forma de hacer esto es sacarle el último elemento a este vectorcito nuestro y explorar el caso donde no se agregue el 3, en ese caso, va a meter al 4. Esa es la importancia del push y el pop. Si lo pensamos desde el principio vamos a tener el caso donde agregamos el 1 y donde no, se van anidando entre sí y nos permiten explorar todo el espacio de soluciones.

b) Calcular la complejidad temporal y espacial del mismo.

El árbol tiene $O(n^k)$ nodos donde en c_{sum} hacen operaciones $O(4)$. Ahora cuando llegamos a las hojas estamos realizando un algoritmo que es $O(k^2)$ con complejidad total $O(n^k + \binom{n}{k} k^2)$

Espacialmente estamos creando n^k vectores de k
por $\rightarrow O(n^k k)$

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

Se le pueden hacer varias podas y mejorar el algoritmo

Una poda por optimalidad sería evitar recorrer

aquella rama cuya suma es menor a uno ya obtenida

→ delinearán recorriendo toda una rama descontando

→ el correcto ya que nos ahorrando expresos computación
→ de 1 elemento