Algoritmos golosos

Santiago Cifuentes

Abril 2023

Plan

Hoy vamos a seguir viendo algoritmos *greedy*, y luego haremos una puesta en común de los resultados del taller. Puntualmente vamos a ver:

- Problema 1: **Cubrimiento de intervalos**.
- Problema 2: **Meximización**.
- Experimentación del problema 2
- Resolución de los ejercicios 1-3 del taller.
- Conclusiones

Cubrimiento

Enunciado

Dado un conjunto $I = \{[s_1, t_1], \ldots, [s_n, t_n]\}$ de intervalos sobre $\mathbb{R}$, queremos encontrar un conjunto P de puntos de tamaño mínimo tal que todo intervalo de I tenga intersección con P.

Si $I = \{[1, 3], [2, 4], [5, 9]\}$, $P = \{3, 7\}$ es un posible óptimo.

Ideas

- Para empezar, ¿Hay algún algoritmo de backtracking que podamos proponer? Es molesto que a priori el conjunto de soluciones posibles es infinito (cualquier subconjunto de puntos de $\mathbb{R}$).
- ¿Podemos restringir el conjunto de puntos a considerar de alguna forma? ¿Podremos suponer que los puntos son siempre bordes de los intervalos?
- Probémoslo.

Mini demo

- Probemos que existe una solución que solo tiene puntos que son bordes de intervalos ¿Cómo lo hacemos?
- Tomemos una solución P con un punto $x \in P$ que no es un borde, y "arreglémosla" ¿Cómo lo hacemos?
- Podemos moverlo hacia la derecha hasta chocar con un borde. Claramente sigue cubriendo los mismos intervalos.
- Observación: ¿Podemos probar algo más fuerte? Podemos suponer que P solo usa los t_i, es la misma demo.

Hacia un greedy

- De ahora en más **vamos a asumir que solo podemos elegir puntos de la forma t_j**. Recién demostramos que esto no nos saca generalidad.
- Con lo que sabemos es fácil hacer un algoritmo de backtracking de complejidad $O(2^n)$, e incluso uno de PD $O(n^2)$. Pero intentemos hacer algo más greedy ¿Ideas?
- Supongamos que están ordenados ¿Qué pasa con el primer t_1? **Tiene** que estar en el conjunto P, ya que todo el resto de los t_i están más adelante.
- ¿Y luego? Podemos buscar el siguiente intervalo (de los que no están cubiertos todavía) con menor t_i, y agregarlo a la lista, bajo el mismo fundamento

Pseudocódigo

$sort(I)$
$P = \{t_1\}$
$ultimo \leftarrow t_1$
for $2 \leq i \leq n$ **do**
    if $s_i > ultimo$ **then**
        $P = P \cup \{t_i\}$
        $ultimo \leftarrow t_i$
    end if
end for
return P

Correctitud

- Demostremos que el algoritmo es correcto ¿De qué forma lo podemos hacer? Demostrando un invariante.
- ¿Como cuál?
- Podemos probar al final de cada iteración i vale $Q(i)$: “P es un conjunto de puntos que cubre a los intervalos 1 a i y está contenido en una solución óptima”.
- Lo probamos por inducción.

Demostración

$Q(i)$: "P es un conjunto de puntos que cubre a los intervalos 1 a i y está contenido en una solución óptima"

- ¿Por qué $Q(1)$ antes de entrar al ciclo? Porque $P = \{t_1\}$, y ya argumentamos que t_1 si o si tiene que estar en la solución.
- Supongamos que vale $Q(i)$, y probemos $Q(i+1)$. Es decir, asumimos que P al principio de la iteración es un conjunto de puntos que cubre a los intervalos 1 a i y se puede extender a una solución P^* (i.e. $P \subseteq P^*$).
- En la iteración $i+1$ pueden pasar dos cosas:
  - ☐ $s_{i+1} \leq ultimo \implies$ En este caso, P ya cubre al intervalo $i+1$. Luego está claro que vale $Q(i+1)$, ya que P es un conjunto de puntos que cubre a los intervalos 1 a $i+1$ que se puede extender a una solución (en particular, a la misma P^* que antes).

Demostración

$Q(i)$: "P es un conjunto de puntos que cubre a los intervalos 1 a i y está contenido en una solución óptima"

- ¿Por qué vale el invariante antes de entrar al ciclo? Porque $P = \{t_1\}$, y ya argumentamos que t_1 si o si tiene que estar en la solución.
- Supongamos que vale $Q(i)$, y probemos $Q(i+1)$. Es decir, asumimos que P al principio de la iteración es un conjunto de puntos que cubre a los intervalos 1 a i y se puede extender a una solución P^* (i.e. $P \subseteq P^*$).
- En la iteración $i+1$ pueden pasar dos cosas:
  - $s_{i+1} > ultimo$ $\Longrightarrow$ En este caso agregamos a P el punto t_{i+1}. Hay que demostrar que esta solución cubre a los primeros $i+1$ intervalos y que es óptima.

Demostración

- ¿Cubre a los primeros $i+1$ intervalos? Si, ya que P cubría a los primeros i, y agregamos un punto para cubrir el $i+1$-ésimo.
- ¿Como vemos que es óptima? ¿Sigue estando contenida en P^*?
- Sea $P^* = P \cup R$. Claramente R tiene que contener un punto t_j que cubra al intervalo $[s_{i+1}, t_{i+1}]$, ya que ningún punto de P cubre a ese intervalo.
- Como están ordenados de menor a mayor, todos los puntos que son distintos a t_{i+1} que quedan tienen que ser mayores o iguales. Luego, la única forma de cubrir este intervalo es usando el punto t_{i+1}.
- Por lo tanto, $t_{i+1} \in R \subseteq P^*$, y luego $P \cup \{t_{i+1}\} \subseteq P^*$.
- Con eso queda probado el invariante.

Conclusión

- El invariante nos garantiza que en la última iteración vale $Q(n)$: "P es un conjunto de puntos que cubre los intervalos 1 a n y se puede extender a una solución óptima".
- Por lo tanto, P es una solución óptima.
- ¿Complejidad del algoritmo? $O(n \log n)$

Meximización

Enunciado (parte 1)

Dado un conjunto de números naturales $X \subseteq \mathbb{N}$ definimos la función $mex : \mathcal{P}(\mathbb{N}) \to \mathbb{N}$ como

$$mex(X) = min\{j | j \in \mathbb{N} \wedge j \notin X\}$$

$mex(\{0, 1, 3\}) = 2$
$mex(\{1, 2, 3, 4, 5\}) = 0$

Meximización

Enunciado (parte 2)

Dado un vector $a = a_1 \ldots a_n$ de números naturales, tenemos que encontrar una permutación $b_1 \ldots b_n$ de a que maximize

$$\sum_{i=1}^{n} mex(\{b_1, \ldots, b_i\})$$

Si $a = \{3, 1, 0\}$, podemos considerar la permutación $\{1, 0, 3\}$, cuyo valor es:

$$mex(\{1\}) + mex(\{1, 0\}) + mex(\{1, 0, 3\}) = 0 + 2 + 2$$

¿Hay una mejor? Sí, $\{0, 1, 3\}$:

$$mex(\{0\}) + mex(\{0, 1\}) + mex(\{0, 1, 3\}) = 1 + 2 + 2$$

Ideas

- ¿Qué hay que hacer para que el primer sumando sea distinto a 0? Hay que tener el 0 en el primer conjunto ¿Qué pasa si 0 no está en a? Todos los sumandos van a dar 0.
- Y si 0 está en a y lo ponemos primero, ¿Qué ponemos como segundo número? Nos gustaría poner un 1 para que el segundo sumando valga 2 ¿Y si no hay un 1?
- En general, queremos que haya una **escalera**. Si a tiene tal forma que la escalera más grande que podemos armar es de altura $k < n$ ¿Qué nos dice eso respecto a los valores de mex? A lo sumo van a ser k. En general, si un valor $x \notin a$ entonces los mex van a estar acotados por x.

Solución golosa

- Vamos a definir nuestra solución b como golosa si empieza con una escalera lo más grande posible.
- Probemos que es óptima ¿Cómo podemos hacer esto? Tomemos una permutación cualquiera p, y mostremos que el valor de b es mayor o igual al valor de p.

Cuentas

- Podemos hacerlos por inducción en la longitud de las permutaciones.
- Es decir, probemos que para $0 \leq j \leq n$ vale $P(j) : \sum_{i=1}^{j} mex(b[1 \dots i]) \geq \sum_{i=1}^{j} mex(p[1 \dots i])$.
- ¿Caso base? $P(0) : \sum_{i=1}^{0} mex(b[1 \dots i]) = 0 = \sum_{i=1}^{0} mex(p[1 \dots i])$.
- Caso recursivo: Alcanza con ver que $mex(b[1 \dots j+1]) \geq mex(p[1 \dots j+1])$. El resto sigue por la hipótesis inductiva.

Cuentas

- Supongamos que la escalera tiene longitud m.
- Si $j+1 \leq m$, ¿Cuánto vale $mex(b[1 \dots j+1])$? Vale $j+1$ ¿Puede el mex de p ser mayor? No, porque el máximo valor posible de un mex de un conjunto de $j+1$ elementos es $j+1$.
- Ahora, si $j+1 > m$, ¿Cuánto vale $mex(b[1 \dots j+1])$? Vale m, ya que la escalera está en el conjunto, y $m \notin a$, y por lo tanto $m \notin b[1 \dots b_{j+1}]$.
- ¿Y qué pasa con $mex(p[1 \dots p_{j+1}])$? Está acotado por m por el mismo motivo. Luego:

$$mex(b[1 \dots b_{j+1}]) = m \geq mex(b[1 \dots p_{j+1}])$$

- Y con esto queda probada la inducción.

Algoritmo

- Ya sabemos que armar una solución con la escalera más grande posible es óptimo. Ahora ¿Cómo lo hacemos?
- Podemos recorrer el vector buscando primero el 0. Luego buscamos el 1, y así hasta encontrar la escalera más grande posible.
- Podemos hacer todo junto: si encontramos un elemento $i < n$ lo ponemos en su posición (i.e. si encontramos un 2, hacemos swap con la posición 2 del vector para ponerlo en su lugar).
- A ver el código.
- ¿Complejidad? Parece $O(n)$, porque un número de la escalera es swapeado a su posición una única vez.

Mini experimento

- Para verificar que la complejidad es efectivamente $O(n)$ vamos a realizar una experimentación.
- Vamos a generar instancias aleatorias de distintos tamaños, y ver cómo va variando la performance del algoritmo.
- ¿Por otro lado, hay instancias que se resuelvan más rápido que otras?
- Las instancias que ya están ordenadas no necesitan swaps. Por otro lado, mientras más swaps haya más lento debería ser el algoritmo.
- Vamos a crear instancias que requieran muchos swaps, y luego compararemos la performance entre ambas familias de instancias.
- Código.

Taller

Taller

Ejercicio 1

Enunciado

Nos dan una lista de n precios de productos que tenemos que comprar. Podemos hacer compras separadas, y por cada compra de k productos nos regalan los $\lfloor \frac{k}{3} \rfloor$ productos más baratos. Queremos maximizar el descuento que recibimos.

Para la lista $\{100, 200, 300\}$ la respuesta es 100.
Para $\{50, 100, 200, 300, 400, 500, 600\}$ es 500.

Ejercicio 1

- Jamás podremos aplicar el descuento sobre los dos más caros, pero si sobre el tercero más caro.
- La propuesta greedy es comprar los productos de a 3, siempre los más caros que quedan.
- ¿Cómo podríamos probar que eso es óptimo?
- Podemos tomar una solución que no arme ese paquete de los 3 más caros, y mostrar que si lo armara obtendríamos una solución mejor.
- Complejidad: $O(n \log(n))$
- Código.

Ejercicio 2

Enunciado

Hay n productos con su precio p_i y su peso w_i. Tenemos G personas que quieren llevarse productos maximizando el precio de lo que se llevan, teniendo en cuenta que cada uno tiene una capacidad c_j. Hay una cantidad infinita de cada objeto.

Ejercicio 2

- Si hubiera una sola persona, ¿Qué problema sería? Knapsack (problema de la mochila)
- ¿Importa que haya más de una? ¿Se afectan entre sí? No, porque las compras de uno no afectan a otro (hay infinitos productos).
- ¿Complejidad de resolver Knapsack? $O(2^n)$ o $O(nW)$ ¿Cuál usamos?
- Complejidad total: $O(NW + G)$.
- Código

Ejercicio 3

Enunciado

Queremos imprimir lexicográficamente todas las cadenas de ceros y unos de longitud N que tengan H unos.

Ejercicio 3

- Como hay que generar todas, seguramente tendremos que hacer algún backtracking. Podemos pensar una recursión que devuelva el conjunto de cadenas:

$$ham(i, h, cur) = \begin{cases} \emptyset \\ \{cur\} \\ ham(i-1, h, cur \oplus 0) \cup ham(i-1, h-1, cur \oplus 1) \end{cases}$$

"Conjunto de cadenas de longitud i que extienden a cur y tienen h unos"

- Llamamos $ham(N, H, \emptyset)$.
- Complejidad: $O(2^N)$, aunque en realidad es $O(\binom{N}{H})$.

Conclusión

Hoy vimos:

- Un algoritmo greedy para el problema de cubrimiento. Tuvimos que demostrar cierta estructura en las soluciones óptimas, y probamos la correctitud del algoritmo con un cierto invariante.
- Un algoritmo para el problema de meximización. En este, directamente probamos que la solución golosa es óptima comparándola con otra arbitraria. (¿Podíamos hacer esto en el problema anterior?)
- Una breve experimentación para corroborar la complejidad teórica y compar algunas familias de instancias.
- Repasamos los ejercicios del taller.

Fin