Algoritmos Online

Elizabeth Ríos Alvarado
Daniel Rojo Mata
Fernando Rodrigo Valenzuela García De León

¿Cómo resolvemos algo cuando no tenemos toda la información?

Opción 1

Probabilidad

Crear un modelo probabilístico de inputs futuros y diseñar un algoritmo que asume los inputs conforme el modelo.

Opción 2

Peor caso

Emplear una estrategia conservadora que limite qué tan mala pueda ser una solución dado cualquier input

Preliminares

Definición: Análisis competitivo

Comparar la solución producida por el algoritmo online con una solución producida por un algoritmo óptimo que conoce los inputs futuros, tomando una proporción del peor caso entre todos los casos posibles.

Definición: Radio competitivo

Sea U el conjunto de todos los posibles inputs, y considérese algún input $I \in U$. Para un problema de minimización, si un algoritmo A produce una solución de valor $A(I)$ en el input I y la solución de un algoritmo F que conoce el futuro tiene valor $F(I)$ para el mismo input, entonces el **radio competitivo** del algoritmo A es

$$\max\left\{\frac{A(I)}{F(I)} : I \in U\right\}$$

Definición: Radio competitivo

Si un algoritmo online tiene un radio competitivo de c, se dice que es **c-competitivo**. El radio competitivo es siempre al menos 1, por lo que se quiere un algoritmo online que tenga su radio competitivo lo más cercano posible a 1.

Agenda

1. El elevador
2. Mantener una lista de búsqueda
3. Online Caching

Problema 1: El elevador

¿Esperar el elevador o tomar las escaleras?

Analizando

Supongamos:

- Por las escaleras, se sube un piso en un minuto.
- De la planta baja al piso k, el elevador se tarda un minuto.
- El número de minutos que le toma al elevador llegar a la planta baja es a lo más $B\text{-}1 \in \mathbb{Z}$, con $B > k$.

Entonces, llegar al piso k...

- Por las escaleras, siempre toma k minutos.
- Por el elevador, lo menos que toma es un minuto.
- Por el elevador, lo más que toma es $(B\text{-}1)+1=B$ minutos.

¿Qué haría una vidente?

Si sabe que el elevador llegará en a lo más *k-1* minutos, esperará por el elevador. De otra forma, tomará las escaleras.

$$t(m) = \begin{cases} m+1 & \text{si } m \leq K-1 \\ K & \text{si } m \geq K \end{cases}$$

El único input es el tiempo que le toma al elevador llegar a la planta baja.

Algoritmo 1

"Siempre irse por las escaleras"

Siempre toma k minutos, por lo que su radio competitivo es el siguiente:

$$\max\left\{\frac{K}{t(m)} : 0 \leq m \leq B-1\right\}$$

$$\max\left\{\frac{K}{1}, \frac{K}{2}, \ldots, \frac{K}{(k-1)}, \frac{K}{K}, \frac{K}{K}, \ldots, \frac{K}{K}\right\}$$

Algoritmo 2

"Siempre irse por el elevador"

Si al elevador le toma m minutos llegar a la planta baja, entonces el algoritmo siempre tarda $m+1$. Entonces, su radio competitivo es:

$$\max\left\{\frac{m+1}{t(m)} \;:\; 0 \le m \le B-1\right\}$$

$$\max\left\{\frac{1}{1}, \frac{2}{2}, \ldots, \frac{K-1}{(k-1)}, \frac{K}{K}, \frac{K+1}{K}, \ldots, \frac{B}{K}\right\}$$

Algoritmo 3

"Esperar un poquito"

Supongamos que "un poquito" es k minutos. Entonces, el tiempo $h(m)$ que se requiere, como función del número m de minutos antes de que llegue el elevador es:

$$h(m) = \begin{cases} m+1 & \text{si } m \leq k \\ 2k & \text{si } m > k \end{cases}$$

Algoritmo 3

"Esperar un poquito"

Entonces, su radio competitivo es:

$$\max\left\{\frac{h(m)}{t(m)} \;:\; 0 \le m \le B-1\right\}$$

$$\max\left\{\frac{1}{1}, \frac{2}{2}, \dots, \frac{K}{K}, \frac{K+1}{K}, \frac{2K}{K}, \dots, \frac{2K}{K}\right\} = 2$$

Ahora es **independiente** de k y de B.

¿Qué algoritmo elegimos?

Algoritmo 3

Porque es el algoritmo que nos protege contra cualquier peor caso

Problema 2: Mantener una lista de búsqueda

Analizando

Supongamos:

- Se tiene una lista doblemente conectada L de n elementos $\{x_1, x_2, ..., x_n\}$
- Denotamos la posición del elemento x_i en la lista L por $r_L(x_i)$, donde $1 \leq r_L(x_i) \leq n$.
- La única forma de reordenar la lista es intercambiar dos elementos adyacentes.
- Llamando al algoritmo `List-Search(L,x_i)`, por lo que tarda $\Theta(r_L(x_i))$.

Si tuviéramos información...

acerca de la distribución de solicitudes de búsqueda, entonces tendría sentido arreglar la lista antes de tiempo para poner al frente los elementos buscados más frecuentemente al frente de la lista, lo cual minimizaría el costo.

Pero como no tenemos información, no importa cómo ordenemos la lista, es posible que todas las búsquedas sean para el elemento que aparece en la cola de la lista.

Move-To-Front

El elemento que se busca, se mueve al frente de la lista.

- Una llamada a M-T-F cuesta $2r_L(k)-1$. Puesto que cuesta $r_L(k)$ buscar a k, y cuesta 1 por cada uno de los $r_L(k)-1$ cambios para mover a k al frente de la lista.
- Tiene un radio competitivo de 4.

Ejemplo

	FORESEE					MOVE-TO-FRONT				
element searched	L	search cost	swap cost	search + swap cost	cumulative cost	L	search cost	swap cost	search + swap cost	cumulative cost
5	$\langle 1, 2, 3, 4, 5 \rangle$	5	0	5	5	$\langle 1, 2, 3, 4, 5 \rangle$	5	4	9	9
3	$\langle 1, 2, 3, 4, 5 \rangle$	3	3	6	11	$\langle 5, 1, 2, 3, 4 \rangle$	4	3	7	16
4	$\langle 4, 1, 2, 3, 5 \rangle$	1	0	1	12	$\langle 3, 5, 1, 2, 4 \rangle$	5	4	9	25
4	$\langle 4, 1, 2, 3, 5 \rangle$	1	0	1	13	$\langle 4, 3, 5, 1, 2 \rangle$	1	0	1	26

Los costos acumulados del algoritmo FORESEE no cambiarían si hubiera movido el 3 al frente después de buscar por 5. Tampoco si 4 se hubiera movido a la segunda posición después de buscar por 5.

Ideas preliminares para la demostración.

Inversión: Dos elementos a y b, en una lista a aparece antes que b y en otra lista b aparece antes que a.

Cuenta de inversiones: Para dos listas L, L', sea $\boldsymbol{I(L,L')}$ el número de inversiones entre esas dos listas.

$\boldsymbol{L_i^M}$ **/** $\boldsymbol{L_i^F}$**:** La lista que obtiene Move-To-Front / Foresee inmediatamente después de la i-ésima búsqueda.

$\boldsymbol{c_i^M}$**/**$\boldsymbol{c_i^F}$**:** El costo ocasionado por Move-To-Front / Foresee en su i-ésima llamada.

Esbozo de la demostración.

1. Se hace un cálculo general de los costos en que incurren tanto Foresee como Move-To-Front.
2. Se usa una función potencial (análisis amortizado) que depende de la cuenta de inversiones. $\Phi_i = 2I(L_i^M, L_i^F)$
3. Se obtiene el costo amortizado de Move-To-Front.
4. Y es menor al costo real de Foresee.

cambio potencial

$$\hat{C}_i^M = C_i^M + \Phi_i - \Phi_{i-1}$$

cuentitas

$$\vdots$$

$$\leq 4C_i^F$$

Esbozo de la demostración.

5. El costo total amortizado da una cota superior al costo total real.
6. De donde se obtiene la siguiente cadena de desigualdades:

$$\sum_{i=1}^{m} c_i^M \leq \sum_{i=1}^{m} \hat{c}_i^M$$

$$\leq \sum_{i=1}^{m} 4c_i^F$$

$$= 4\sum_{i=1}^{m} c_i^F$$

Esbozo de la demostración.

7. Por lo tanto, el costo total de m operaciones de Move-To-Front es a lo más 4 veces el costo total de m operaciones de Foresee.
8. Move-To-Front es 4-competitivo.

■

Problema 3: Online Caching

Analizando

- El input comprende una secuencia de n solicitudes de memoria, para data en bloques b_1, b_2, ..., b_n, en ese orden.
- Después de que el bloque b_i es solicitado, queda almacenado en una caché que puede mantener hasta k bloques.
- ¿Qué datos mantener en la memoria rápida y pequeña para minimizar los fallos de caché?
- Si el bloque solicitado no está en la caché y ésta está llena, el algoritmo debe decidir qué bloque reemplazar para liberar espacio.

Supongamos:

- La caché comienza vacía, por lo que no ocurren expulsiones durante las primeras k solicitudes.
- $n > k$, puesto que de otra forma no existiría el problema.
- Podrían no serlo, pero supondremos que al menos son solicitados k bloques distintos.

Algoritmos deterministas

LIFO

Last In, First Out

Reemplaza el bloque más reciente, lo cual puede ser ineficiente porque estos bloques podrían necesitarse pronto.

Tiene un radio competitivo de: $\Theta(n/k)$

Esbozo de la demostración.

Cota inferior.

1. Supongamos que el input consiste en $k+1$ bloques tal que la secuencia de solicitudes es: $1, 2, ..., k, k+1, k, k+1, k, k+1, ...$
2. El algoritmo se cicla en expulsar e insertar los bloques k y $k+1$, lo cual genera un fallo en cada una de las n solicitudes.
3. Un algoritmo offline sabría la secuencia de solicitudes y después de la primera solicitud del bloque $k+1$ expulsaría cualquier otro bloque que no fuera ni k, ni $k+1$. Dado que las primeras k solicitudes se consideran fallos, el total de fallos sería $k+1$.
4. El radio competitivo de LIFO es al menos $n / (k+1)$ o $o(n/k)$.

Esbozo de la demostración.

Cota superior.

1. Notemos que para cualquier input de tamaño n, cualquier algoritmo fallaría a lo más n veces.
2. Dado que el input contiene al menos k bloques distintos, cualquier algoritmo (incluido el algoritmo offline óptimo) se equivoca al menos k veces.
3. Por lo tanto, LIFO tiene un radio competitivo de a lo más $O(n/k)$

LRU

Least Recently Used

Reemplaza el bloque que ha estado más tiempo sin ser usado, asumiendo que los bloques usados recientemente serán solicitados pronto.

Tiene un radio competitivo de: $\Theta(k)$

Esbozo de la demostración.

1. Dividimos la secuencia de solicitudes entre épocas.
   a. La época 1 comienza con la primera solicitud
   b. La época i, para $i > 1$, comienza al encontrarse con la $(k+1)$-ésima solicitud distinta desde el comienzo de la época $i-1$.
2. Sólo la primera solicitud de un bloque dentro de una época puede provocar una falla de caché y la caché contiene k bloques, cada época incurre en k fallas de caché como máximo.
3. El algoritmo óptimo, para cada época, incurre en al menos una falla de caché por definición de las épocas.
4. El radio competitivo es a lo más $k/1 = O(k)$.

FIFO

First In, First Out

Reemplaza el bloque que lleva más tiempo en la caché, pero no es siempre eficiente, ya que no considera la frecuencia de acceso.

Tiene un radio competitivo de: $\Theta(k)$

LFU

Least Frequently Used

Reemplaza el bloque menos usado históricamente, asumiendo que los bloques menos solicitados en el pasado serán menos necesarios en futuras consultas.

Tiene un radio competitivo de: $\Theta(n/k)$

Algoritmos aleatorizados

Recordemos...

que analizamos los algoritmos online con respecto a un adversario que conoce el algoritmo online y puede diseñar solicitudes conociendo las decisiones que toma. Con la aleatorización, debemos preguntarnos si el adversario también conoce las elecciones aleatorias que realiza el algoritmo en línea.

- Adversario ignorante: No conoce las elecciones.
- Adversario conocedor: Conoce las elecciones.

El algoritmo

```
RANDOMIZED-MARKING(b)
1  if block b resides in the cache,
2       b.mark = 1
3  else
4       if all blocks b' in the cache have b'.mark = 1
5            unmark all blocks b' in the cache, setting b'.mark = 0
6       select an unmarked block u with u.mark = 0 uniformly at random
7       evict block u
8       place block b into the cache
9       b.mark = 1
```

Esbozo de la demostración.

1. La secuencia de solicitudes se divide en épocas; cada época tiene solicitudes nuevas y viejas, donde solo las nuevas siempre causan fallos de caché.
2. La probabilidad de fallo en una solicitud vieja depende de si el bloque fue expulsado previamente en la misma época.
3. El número total de fallos en una época se calcula sumando los fallos de solicitudes nuevas y viejas, siendo siempre un fallo en cada solicitud nueva.
4. Para el algoritmo `Randomized-Marking`, el número esperado de fallos en caché en una época se acota superiormente, según la estructura probabilística de los fallos en solicitudes viejas.
5. Comparando este número de fallos con el mínimo posible para un algoritmo óptimo, se concluye que la competitividad del algoritmo es del orden $O(\log k)$.

Conclusiones

Estos algoritmos son necesarios para diferentes problemas que reciben datos de manera progresiva y se requiere tomar decisiones en tiempo real. Estas decisiones son tomadas sin conocer la secuencia futura de entradas.

Gracias al análisis competitivo, podemos ver como estos algoritmos no pueden igualar un algoritmo óptimo que conoce todo el conjunto de datos, pero sí pueden mantener una eficiencia razonable en diferentes contextos, así estimamos el peor caso y elegimos estrategias que reduzcan las desventajas que nacen al trabajar sin todo el conjunto de entradas o todo el conocimiento del problema.