Algoritmos Online

ELIZABETH RÍOS ALVARADO, UNAM, México
DANIEL ROJO MATA, UNAM, México
FERNANDO RODRIGO VALENZUELA GARCÍA DE LEÓN, UNAM, México

Los algoritmos en línea son herramientas fundamentales para resolver problemas en entornos donde la información se revela de forma secuencial y no se conoce el futuro. Estos algoritmos deben tomar decisiones óptimas en tiempo real, basándose únicamente en los datos disponibles hasta ese momento. El presente trabajo introduce los conceptos básicos de los algoritmos en línea y su análisis competitivo, que permite evaluar su desempeño en comparación con un algoritmo óptimo que conoce el futuro. Se presentan ejemplos clásicos como el problema del elevador, la reorganización de listas y el problema de caché, ilustrando la importancia de estos algoritmos en diversas situaciones. Además, se define el radio competitivo como una métrica clave para medir la eficiencia de un algoritmo en línea

Additional Key Words and Phrases: algoritmos en línea, análisis competitivo, radio competitivo, problema del elevador, reorganización de listas, caché, algoritmos aleatorizados.

1. INTRODUCCIÓN

Los algoritmos en línea, también conocidos como *online algorithms*, son aquellos que deben tomar decisiones basándose únicamente en la información disponible hasta el momento, sin conocer el futuro. Esto significa que, a diferencia de los algoritmos fuera de línea (*offline algorithms*), que pueden trabajar con todos los datos previamente, los algoritmos en línea deben adaptarse a información progresiva, tomando decisiones en tiempo real y sin un conocimiento completo del ambiente.

De esta forma, se pueden imaginar varios escenarios donde no se conocen todos los datos necesarios pero sí se requiere una respuesta inmediata. Ejemplos de esto incluyen la gestión de inventario en una tienda grande o la asignación de recursos en sistemas computacionales, de esta manera, nacen estos algoritmos para poder tomar mejores decisiones a falta de conocimiento de una situación.

Para evaluar su efectividad, se realiza *análisis competitivo*, cuya finalidad es medir la eficiencia de un algoritmo en línea, dándonos una "garantía" de cuán cerca puede estar de la solución óptima en el peor de los casos.

2. PRELIMINARES

Para comprender mejor el funcionamiento de estos algoritmos, se hace el estudio de tres problemas en los cuales se usa de manera natural las características de los algoritmos online.

Considere el llamado *problema del elevador*. El objetivo de este problema es decidir si esperar el elevador o subir las escaleras, con el fin de llegar lo antes posible a un piso deseado. Como no se sabe exactamente dónde está el elevador, y subir las escaleras siempre lleva el mismo tiempo, en general sería mejor elegir las escaleras para evitar el riesgo de que el elevador se demore. Sin embargo, si se decide esperar un tiempo específico antes de optar por las escaleras, el tiempo ya no dependería de la velocidad del elevador o del esfuerzo al subir, sino del lapso que uno esté dispuesto a esperar antes de tomar acción.

Authors' addresses: Elizabeth Ríos Alvarado, UNAM, CDMX, México, elizabethra@ciencias.unam.mx; Daniel Rojo Mata, UNAM, CDMX, México, danielrojomata@gmail.com; Fernando Rodrigo Valenzuela García de León, UNAM, CDMX, México, ferrodrivalenzuela@gmail.com.

Otro ejemplo en el contexto de computación, sería un problema relacionado con listas. Aquí, un algoritmo en línea debe decidir cómo reorganizar una lista tras cada búsqueda, con el fin de minimizar el tiempo total de búsqueda sin conocer el orden de futuras búsquedas. Como se desconoce la secuencia de las búsquedas futuras, el algoritmo debe decidir una estrategia para reorganizar los elementos de manera eficiente sin ser innecesario. Un algoritmo online debe de ser lo suficientemente flexible para reorganizar la lista después de cada búsqueda, de manera que la lista se pueda adaptar de manera gradual.
Un algoritmo común para esto es *MOVE-TO-FRONT*, que mueve al inicio de la lista los elementos buscados, aumentando la probabilidad de encontrar rápidamente elementos que se utilizan frecuentemente en consultas sucesivas.

Un problema que se puede considerar clásico dentro de los algoritmos en línea es el llamado *Online Caching*, haciendo referencia a la memoria caché que tienen las computadoras. Este problema se refiere a cómo seleccionar datos para mantener en la memoria caché de una computadora, minimizando la necesidad de acceder a la memoria principal, que es generalmente más lenta. Tiene su raíz en algoritmos en línea porque se desconocen las futuras solicitudes que se harán.

Por ejemplo, si la solicitud se encuentra en el caché, no es necesario realizar ninguna acción, sin embargo, si la solicitud no se encuentra en el caché, el algoritmo debe elegir cuál de los bloques actuales en el caché reemplazar para hacer espacio, intentando minimizar los fallos en solicitudes futuras.

Al no conocer las siguientes secuencias, se deben tomar decisiones basadas solo en las solicitudes, por lo cual se aplicarán las políticas de reemplazo de caché determinista (FIFO, LIFO, LRU y LFU).

También se puede aplicar la aleatoriedad en la toma de decisiones de reemplazo en el caché. Un *algoritmo aleatorizado* no sigue una regla fija para elegir qué bloque reemplazar, sino que lo hace de manera probabilística. Esto reduce la posibilidad de que un adversario manipule la secuencia de solicitudes para aumentar los fallos. Un ejemplo de este enfoque es el algoritmo *RANDOMIZED-MARKING*, que utiliza aleatoriedad para decidir cuál bloque reemplazar.

2.1. Definiciones principales

Sea $\mathcal{U}$ el conjunto de todos los posibles inputs, y considérese algún input $I \in \mathcal{U}$. 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:

$$max\{A(I)/F(I) : I \in \mathcal{U}\} \tag{1}$$

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.

3. EXPLICACIÓN DE LOS ALGORITMOS

3.1. Problema del Elevador

Alguien entra a un edificio y necesita llegar a una oficina que está k pisos arriba. Tiene dos opciones: subir por las escaleras o usar el elevador. Suponga, para simplificar, que puede subir por las escaleras a un ritmo de un piso por minuto. El elevador es mucho más rápido: puede subir los k pisos en solo un minuto. Su dilema es que desconoce cuánto tiempo tardará el elevador en llegar a la planta baja para recogerlo. ¿Le conviene tomar el elevador o las escaleras? ¿Cómo debería

decidir?
Subir por las escaleras siempre toma k minutos. Si el elevador puede tardar hasta $B-1$ minutos en llegar, siendo B considerablemente mayor que k, ¿es mejor esperar el elevador o tomar las escaleras? Se puede utilizar un análisis competitivo para orientar la decisión sobre si tomar las escaleras o esperar el elevador.
Considere primero lo que haría una vidente[1] en esta situación. Si la vidente sabe que el elevador llegará en un máximo de $k-1$ minutos, entonces optará por esperar el elevador, de otra forma, la vidente decidirá tomar las escaleras. Sea m el número de minutos que tarda el elevador en llegar al primer piso. Se puede expresar el tiempo total como la siguiente función:

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

Considere los siguientes algoritmos para la solución del problema.

- **Siempre tomar las escaleras**. El radio competitivo, ecuación 1, es de la forma:

$$\text{máx}\left\{\frac{k}{t(m)} : 0 \leq m \leq B-1\right\} = \text{máx}\left\{\frac{k}{1}, \frac{k}{2}, \ldots, \frac{k}{k-1}, \frac{k}{k}, \frac{k}{k}, \ldots, \frac{k}{k}\right\} = k$$

- **Siempre tomar el elevador**. Si al elevador siempre le toma m minutos el llegar a la planta baja, entonces el algoritmo tomará $m+1$ minutos.
  El radio competitivo, ecuación 1, es:

$$\text{máx}\left\{\frac{m+1}{t(m)} : 0 \leq m \leq B-1\right\} = \text{máx}\left\{\frac{1}{1}, \frac{2}{2}, \ldots, \frac{k}{k}, \frac{k+1}{k}, \frac{k+2}{k}, \ldots, \frac{B}{k}\right\} = \frac{B}{k}$$

- **Esperar un poco**. Esperar un tiempo de k minutos para ver si el elevador llega. Si el elevador no llega en ese tiempo, se toman las escaleras para no seguir perdiendo más tiempo.
  Matemáticamente hablando:

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

  La razón de tener $2k$ es porque se espera k minutos y después se sube las escaleras por k minutos. El radio competitivo, ecuación 1, es:

$$\text{máx}\left\{\frac{h(m)}{t(m)} : 0 \leq m \leq B-1\right\} = \text{máx}\left\{\frac{1}{1}, \frac{2}{2}, \ldots, \frac{k}{k}, \frac{k+1}{k}, \frac{2k}{k}, \ldots, \frac{2k}{k}\right\} = 2$$

Con base a los resultados anteriores se concluye que *siempre tomar las escaleras* tiene un radio competitivo k, lo cual es una opción segura, pero ineficiente si el elevador llega rápidamente. *Siempre tomar el elevador* tiene un radio competitivo $\frac{B}{k}$; funciona si el elevador llega rápido, pero puede ser muy costoso en tiempo si tarda mucho. Por otro lado, *esperar un poco* tiene un radio competitivo de 2; balancea ambos extremos, limitando el tiempo en el peor de los casos y siendo ésta la mejor opción.

3.2. Mantenimiento de Lista

Sea L una lista doblemente enlazada de n elementos $\{x_1, x_2, \ldots, x_n\}$. El objetivo es reorganizar L después de cada consulta para minimizar el tiempo total de búsqueda, sin conocer el orden de las consultas futuras.
Se denota a la posición del elemento x_i en la lista L por $r_L(x_i)$, donde $1 \leq r_L(x_i) \leq n$. Cada llamada al algoritmo `List-Search`(L, x_i) toma $\Theta(r_L(x_i))$.

[1] Hace mención a un algoritmo *offline* que sea óptimo

Supóngase también que la única forma de reordenar la lista es intercambiando dos elementos adyacentes, con un costo 1 para cada operación.

Si se tuviera información de antemano acerca de la distribución de las solicitudes de búsqueda, entonces tendría sentido acomodar la lista antes de tiempo con los elementos más buscados al frente, lo cual minimizaría el costo.

Dado que no se tiene dicha información, no importa cómo se ordene la lista, es posible que cada búsqueda sea por el elemento que aparece al final de la lista. El tiempo total de búsqueda sería $\Theta(nm)$, donde m es el número de búsquedas.

En lugar de sólo evaluar rendimiento en la peor secuencia posible, se comparará una estrategia de reorganización con lo que haría un algoritmo offline óptimo si supiera la secuencia de búsqueda por adelantado.

3.2.1. MOVE-TO-FRONT. Este algoritmo ataca directamente el problema de la lista funcionando de la siguiente forma:

- **Busca un elemento:** se busca el elemento x en la lista. Esto requiere. $O(r_L(x))$ operaciones, donde $r_L(x)$ es la posición de x antes de iniciar la búsqueda.
- **Reubicar elemento:** Después de encontrar el elemento buscado, este se mueve al inicio de la lista, haciendo un intercambio de posiciones, costando: $O(r_L(x)) - 1$

El fin de este algoritmo es que si un elemento se vuelve a buscar, estará al inicio, reduciendo significativamente el tiempo necesario para buscarlo.

Este algoritmo tiene un radio competitivo de 4.

Se hará uso de una función potencial que depende de la cuenta de inversiones, mide el cambio en el costo causado por el número de inversiones entre el algoritmo Move-To-Front y el algoritmo óptimo offline (llamado Foresee).

Para cada operación de Move-To-Front, el costo amortizado $\hat{c}_i^M$ se calcula sumando el costo real de la operación con el cambio en el potencial. Dado que cada operación de Move-To-Front aumenta el costo de traer un elemento al frente en función del número de inversiones, el potencial también cambia según las alteraciones en las listas de ambos algoritmos.

Finalmente, se establece la siguiente desigualdad:

$$\begin{aligned}\sum_{i=1}^{m} c_i^M &\leq \sum_{i=1}^{m} \hat{c}_i^M \\ &\leq \sum_{i=1}^{m} 4c_i^F \\ &= 4\sum_{i=1}^{m} c_i^F\end{aligned}$$

Por lo tanto, el algoritmo Move-To-Front es 4-competitivo.

3.3. Gestión de la memoria caché

La gestión de la memoria caché implica decidir qué datos mantener en una memoria rápida y pequeña para minimizar los *fallos de caché* (cuando los datos solicitados no están en la caché y deben buscarse en la memoria principal).

Hay dos enfoques para resolver el problema:

1. **Versión offline:** Se conoce toda la secuencia de solicitudes de datos, lo que permite decidir óptimamente qué reemplazar y minimizar los fallos.
2. **Versión online:** Las solicitudes llegan sin aviso, y el algoritmo debe tomar decisiones en el momento, sin conocer solicitudes futuras, lo que hace más difícil optimizar el rendimiento.

El objetivo es minimizar los fallos de caché al procesar cada solicitud. Si el bloque solicitado no está en la caché y ésta está llena, el algoritmo debe decidir qué bloque reemplazar para liberar espacio.

Existen algunos algoritmos muy comunes para poder determinar qué bloque evitar. Se mencionan dos de éstos y se esboza la demostración de la complejidad del radio competitivo.

3.3.1. LIFO (Last In, First Out). Reemplaza el bloque más reciente, lo cual puede ser ineficiente porque estos bloques podrían necesitarse pronto.

Este algoritmo tiene un radio competitivo de orden $\Theta\left(\frac{n}{k}\right)$.

Suponga que la entrada consiste en $k+1$ bloques, numerados como $1, 2, \ldots, k+1$, y la secuencia de solicitudes es:

$$1, 2, \ldots, k, k+1, k, k+1, k, k+1, \ldots$$

donde, después de la secuencia inicial $1, 2, \ldots, k, k+1$, el resto de la secuencia alterna entre k y $k+1$, con un total de n solicitudes. El algoritmo entra en un ciclo de expulsar y volver a insertar los bloques k y $k+1$, generando un fallo en solicitudes posteriores a la k-ésima, mientras que un algoritmo óptimo solo necesitaría expulsar un bloque una vez, resultando en muchos menos fallos. El radio competitivo de LIFO es, entonces, al menos $\Omega\left(\frac{n}{k}\right)$. Por otra parte, cualquier algoritmo de caché incurre en a lo más n fallos de caché debido a que hay al menos k bloques distintos en la entrada de tamaño n, por lo que es cierto que $O\left(\frac{n}{k}\right)$.
Por lo tanto, el algoritmo LIFO tiene un ratio competitivo de $\Theta\left(\frac{n}{k}\right)$.

3.3.2. LRU (Least Recently Used). Reemplaza el bloque que ha estado más tiempo sin ser usado, asumiendo que los bloques usados recientemente serán solicitados pronto.

El algoritmo tiene un radio competitivo de orden $O(k)$, teniendo n solicitudes y un tamaño de caché de orden k.

Se divide la secuencia de solicitudes en "épocas". Cada época comienza cuando se solicita el $(k+1)$-ésimo bloque distinto desde el inicio de la época anterior. Esto asegura que cada época incluye al menos $k+1$ bloques distintos, donde k es el tamaño de la caché.

En cada época, LRU puede tener como máximo k fallos de caché. Esto se debe a que solo los primeros accesos a cada uno de los $k+1$ bloques distintos en la época pueden causar fallos. Una vez que un bloque se ha cargado en la caché, cualquier solicitud adicional para ese bloque dentro de la misma época no causa fallos, ya que sigue siendo uno de los k bloques más recientemente usados. El algoritmo óptimo tiene al menos 1 fallo de caché por época, ya que, por definición de cada época, siempre hay un bloque que no ha sido solicitado en las k solicitudes previas y, por lo tanto, debe ser cargado en la caché.
Al comparar los fallos de LRU y los del algoritmo óptimo por época, se obtiene el radio competitivo; $O(k)$, lo que significa que LRU puede tener, en el peor caso, k veces más fallos que el algoritmo óptimo.

3.4. Randomized-Marking

Cuando se habla de algoritmos en línea, se consideran situaciones donde las decisiones deben tomarse sin conocer el futuro. La *aleatorización* introduce un elemento de azar en estos algoritmos, lo que puede mejorar su rendimiento en ciertos casos. Para ello, es necesario tener en consideración el papel que juegan los llamados *adeversarios*.
Se consideran dos tipos de adversarios: el *no ignorante*, que conoce el algoritmo y las decisiones aleatorias, y el *ignorante*, quien no las conoce. El adversario indiferente es más débil, pero sigue siendo un reto. Un *mayor poder de la aleatorización* es posible porque el adversario ignorante no pueda anticipar las decisiones aleatorias, limitando su capacidad para diseñar solicitudes adversarias.

Los algoritmos que funcionan bien contra adversarios ignorantes suelen ser más *robustos* y tener mejor rendimiento promedio.

En este contexto, *MARKING* es un algoritmo de reemplazo de caché que marca los bloques recientemente usados. Cuando todos los bloques están marcados y ocurre una solicitud que no está en la caché, el algoritmo desmarca todos los bloques y selecciona uno sin marcar para expulsarlo y hacer espacio. Para elegir este bloque a expulsar entre los bloques sin marcar, el procedimiento *RANDOMIZED-MARKING* lo selecciona de manera aleatoria, tomando como entrada el bloque solicitado. El algoritmo se muestra a continuación:

```
# RANDOMIZED-MARKING(b)
if b está en la caché:
    marcar b como 1
else:
    if todos los bloques b' en la caché tienen b'.mark == 1:
        desmarcar todos los bloques b' en la caché, estableciendo b'.mark = 0
    seleccionar un bloque no marcado u con u.mark == 0, de manera uniforme al azar
    expulsar el bloque u
    colocar el bloque b en la caché y marcarlo como 1
```

El radio competitivo de este algoritmo es del orden de $O(\log k)$, se hace un esbozo de la demostración.

El algoritmo debe decidir qué bloques mantener en caché cuando recibe una secuencia de solicitudes de bloques. El algoritmo se evalúa frente a un adversario *indiferente*.

El análisis se divide en *épocas*: cada época tiene exactamente k solicitudes (excepto la última, que puede tener menos). Dentro de cada época, las solicitudes pueden ser *nuevas* (bloque solicitado por primera vez en la época) o *viejas* (bloques solicitados en la época anterior). Las solicitudes nuevas siempre causan un *fallo de caché*, ya que el bloque no está en la caché al inicio de la época. Las solicitudes viejas pueden causar un fallo de caché si el bloque fue expulsado previamente por una solicitud nueva.

La probabilidad de que una solicitud vieja cause un fallo depende de si el bloque solicitado fue desalojado por alguna solicitud nueva anterior en la misma época. Utilizando un enfoque probabilístico (tipo *bolas y urnas*), se calcula que la probabilidad de un fallo para la j-ésima solicitud vieja es:

$$\frac{n_{ij}}{k-j+1},$$

donde n_{ij} es el número de solicitudes nuevas que ocurrieron antes de la j-ésima solicitud vieja.

Se definen variables aleatorias para contar los fallos de caché. La cantidad total de fallos en la época i es X_i, y su valor esperado se calcula sumando las esperanzas de los fallos por solicitud nueva y vieja, siendo r_i el número de solicitudes nuevas durante la época i:

$$X_i = \sum_{j=1}^{r_i} Y_{ij} + \sum_{j=1}^{r_i} Z_{ij},$$

donde Y_{ij} es el indicador de un fallo de caché para la j-ésima solicitud vieja, y Z_{ij} para la j-ésima solicitud nueva. Como las solicitudes nuevas siempre causan un fallo de caché, se tiene $Z_{ij} = 1$.

El valor esperado de X_i se calcula como:

$$E[X_i] = \sum_{j=1}^{r_i} E[Y_{ij}] + \sum_{j=1}^{r_i} E[Z_{ij}],$$

utilizando la linealidad de la expectativa y las probabilidades calculadas anteriormente.

El número esperado de fallos en caché para el algoritmo `RANDOMIZED-MARKING` se acota superiormente por:

$$r_i \cdot H_k,^{2}$$

donde H_k es el k-ésimo número armónico. Esto se debe a la estructura probabilística de los fallos en las solicitudes viejas.

Para el algoritmo óptimo offline, que conoce todas las solicitudes futuras, no puede haber menos de r_i fallos en la caché en cada época, ya que debe haber al menos r_i bloques nuevos que no estaban en la caché antes.

El número total de fallos en caché del algoritmo óptimo offline es:

$$\sum_{i=1}^{p} m_i \geq \sum_{i=1}^{p} r_i,$$

donde m_i es el número de fallos en la época i para el algoritmo offline.

El radio competitivo es entonces:

$$\frac{E[X]}{\sum_{i=1}^{p} r_i}.$$

Dado que el número esperado de fallos de `RANDOMIZED-MARKING` está acotado por $H_k \sum_{i=1}^{p} r_i$ y el número de fallos del algoritmo offline está acotado inferiormente por $\sum_{i=1}^{p} r_i$, el radio competitivo es:

$$\begin{aligned}
\frac{E[X]}{\sum_{i=1}^{p} r_i} &= \frac{H_k \sum_{i=1}^{p} r_i}{\frac{1}{2} \sum_{i=1}^{p} r_i} \\
&= 2H_k \\
&= 2\ln(k) + O(1) \\
&\in O(\log k)
\end{aligned}$$

Por lo que el radio competitivo es del orden de:

$$O(\log k).$$

[2] Para enteros positivos n, el n-ésimo número armónico es:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1).$$

4. 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, se puede 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 estimando el peor caso y eligiendo estrategias que reduzcan las desventajas que nacen al trabajar sin todo el conjunto de entradas o todo el conocimiento del problema.

Con todo esto, se aprecia que los algoritmos en línea han marcado un avance importante para la toma de decisiones dentro de problemas existentes relacionados con situaciones reales, tales como el manejo de la memoria caché, el cual ayuda a comprender mejor el manejo de la información que existe dentro de un equipo de cómputo.

Received 05 Noviembre 2024; revised 11 Noviembre 2024; accepted 13 Noviembre 2024