

CacheOpt

19. Como los accesos a memoria RAM son lentos en comparación al trabajo propio del CPU, es común que se coloquen memorias intermedias más diminutas y de alta velocidad entre ambas unidades, las cuales llamamos *cachés*. Cuando un programa se ejecuta y hace una consulta a la memoria por cierta posición r primero se verifica si la posición r está cargada en la caché, en cuyo caso el CPU la puede obtener sin tener que hacer el acceso a la RAM. Cuando esto ocurre, decimos que ocurre un *caché hit*. En cambio, si la posición r no está en la caché, esta se busca a memoria, se carga en la caché, y luego se la informa al CPU. A este evento se lo conoce como *caché miss*.

Como la caché es más chica que la memoria RAM es inevitable que eventualmente ocurra un *caché miss* y que la caché esté llena. En ese caso, la caché debe decidir qué información va a desechar para darle lugar a la nueva entrada. Naturalmente, se busca minimizar la cantidad de *misses* de los siguientes accesos.

El problema de *Off-line caching* consiste en determinar, dada una caché C de tamaño k y una lista de n requests $R = \{r_1, r_2, \dots, r_n\}$ ² a posiciones de memoria, qué decisión debe tomar en cada paso la caché para minimizar la cantidad de *misses*. Por ejemplo, si $k = 2$ y $R = \{1, 2, 3, 1\}$ entonces:

- La primera consulta es un *miss*, pero como hay lugar en la caché (empieza vacía) se carga la posición 1 a C ($C = \{1\}$).
- Con la segunda consulta pasa lo mismo, por lo que la caché queda en el estado $C = \{1, 2\}$.
- En la tercera consulta la caché está llena, por lo que se debe desechar alguna entrada. Notemos que si se desecha 1 entonces la cuarta consulta dará otro *miss*, mientras que si se desecha 2 entonces habrá un *hit*.

Una política posible para decidir qué elemento desechar es la *furthest-in-future*: se desecha aquella posición r cuya siguiente acceso es el más lejano (o bien, que no tiene un siguiente acceso).

- a) **Opcional:** Definir una función recursiva $f(i, mem)$ que tome un índice y un estado de la memoria y devuelva la mínima cantidad de *caché misses* que deben ocurrir para procesar todas las consultas $\{r_i, r_{i+1}, \dots, r_n\}$ si el estado actual de la memoria es mem . ¿Con qué llamado se resuelve el problema? Estudiar la superposición de subproblemas y explicar en qué casos vale la pena memorizar.
- b) Probar que la política *furthest-in-future* es óptima (es decir, que minimiza la cantidad de *misses*). **Ayuda:** Dada una serie de decisiones, probar que si en un paso no se sigue la política *furthest* entonces podemos alterar ese paso para que sí la siga sin afectar la cantidad de *misses*.
- c) Dar un algoritmo con complejidad temporal $O(n \log(k))$ que informe qué decisión debe tomar la caché en cada paso para minimizar la cantidad de *misses*.

③ Supongamos $R = \{1, 2, 3, 4, 3, 2, 6, 7, 1\}$ y $k = 3$

Para $i=0$, $R_i = \text{miss}$ y $k=2$ $C=1$

Para $i=1$ $R_i = \text{hit}$ y $k=2$ $C=1$

$i=2$ $R_i = \text{miss}$ $k=1$ $C=1, 2$

$i=3$ $R_i = \text{miss}$ $k=0$ $C=1, 2, 3$

$i=4$ $R_i = \text{miss}$ \rightarrow de quién me deshago? Del 1
C

i-5 R_i miss \Rightarrow me ducho del 5

La idea algorítmica va a ser: una vez $k=0$ fijame el elemento del cache que más tarde de responderá. Si alguno de mis elementos está a menos de k por de responder no lo saco del cache.

Idea algorítmica.

1. Recorro R_i y armo un diccionario donde considero como clave R_i y su significado una lista enlazada con las posiciones $O(n)$
2. Lleno mi cache C con los primeros k elementos no repetidos de R_i . A medida que voy viendo si es un hit o un miss borro la primera posición de mi lista.
3. A c/pas del cache le aviso

CacheOpt (R , k)

Dicc Vars \rightarrow (clave, significado) = (r_i , lugares donde aparece) $O(1)$

for (int $i=0$; $i < |R|$; $i+1$) $\{ O(n)$ con $n=|R|$

if !def($D[R[i]]$) then $O(1)$
definir($R[i]$) D $O(1)$

endif

agregar (i) D $O(1)$
endfor

H = more_heap (modo) $O(1)$

misses = 0; $O(1)$

P = dicc Vars \rightarrow (clave, significado) = (r_i , ultApEnElHeap) $O(1)$

for (int $j=0$; $j < |R|$; $j+1$) $\{ O(n)$

if $D[R[j]] = \text{value}$ then $O(1)$

: posAp = $|R| + 1 \rightarrow$ no trae la aparece $O(1)$

else

: posAp = $\max(D[R[j]])$ $O(1)$

endif

if !def($P[R[j]]$) then $O(1)$

H.eliminar_pos($P[R[j]]$) $O(\log k)$

insertar_posElHeap $O(1)$

H.agregar ($R[j]$, posAp, posElHeap) $O(\log k)$

$P[R[j]] = \text{posElHeap}$ $O(1)$

eliminar $\{ D[R[j]]\}$ $O(1)$

else

misses++; $O(1)$

if H.size $\neq k$ then

eliminar $\{ D[R[j]]\}$ $O(1)$

insertar_posElHeap $O(1)$

tupla = $(R[j], \text{1}^{\text{ra}} \text{ pos. } D[R[j]] \text{ por el heap})$ $O(1)$
 $\Theta(\log k)$ $\Pi.$ agraga(tupla) \rightarrow la prioridad la determina la 2^{da} pos de la tupla =
 $P(R[j]) = \text{pos en el heap}$ $O(1)$ $\text{ultimo ap de la insercción}$
else

$\Pi.$ EliminarMax $O(\log k)$

eliminar 1^{ra} $D[R[j]]$ $O(1)$

poner en el heap $O(1)$

tupla = $(R[j], \text{1}^{\text{ra}} \text{ pos. } D[R[j]] \text{ por el heap})$ $O(1)$

$\Pi.$ agraga(tupla) $O(\log k)$

$P(R[j]) = \text{pos en el heap}$ $O(1)$

endif

endif

endifor

return minse $O(1)$

Complejidad temporal: $O(n \cdot \log k)$
espacio: $O(n)$