演算法程式作業三

Offline Caching Problem

110502518陳文獻

題目說明

- 目前系統有大小為 k 個 block 的 cache C，資料請求序列為 $b1, b2, \dots, bN$。使用 Furthest-in-future 的機制管理，對於每一筆 request， 若 cache hit 輸出 “hit”，cache miss 輸出 “miss”，其中若有將 block 移 出則在 “miss”的下一行輸出 “evict x“ x 為移出的元素。如 cache 內有多個未來都不會用到的 block，則優先移出先進入到 cache 的 block。

Cache 管理資料結構說明

- sequence[N] := 填入輸入序列的串列
- futures[N] := futures[i] 紀錄sequence在i後下一個出現 sequence[i] block 的位置

  (若之後都不出現，future[i]設為大整數，其中i越小future越大)
- 大小K的priority queue 紅黑樹 C := 相當cache， key是該block下次出現的位置 I, value是block編號
- 大小K的hash map Inverse := 與C反向對應的hash map
- 大小K的hash map M := 預處理時方便用的資料結構，紀錄block上次出現的位置

Pseudo Code 與演算法說明

1. 輸入sequence[N] O(N)
2. 利用sequence[N] 建立futures[N] O(N)
    1. 初始化futures = [-1, -1, …, -1]; BIGINT = 2 * N + 1
    2. for i from 0 to N-1:
    3. 	if sequence[i] in m:
    4. 		futures[m[item]] = i;
    5. 	m[item] = i;
    6. for i from 0 to N-1:
    7. 	if (futures[i] == -1)
    8. 	 futures[i] = BIGINT;
    9. 	 BIGINT -= 1

Pseudo Code 與演算法說明

3. 再掃描一次sequence模擬cache的行為 O(N log K)

```
1.  for i from 0 to N-1:
2.      if sequence[i] in Inverse:
3.          print “hit”
4.          remove (i, sequence[i])                    O(log K)
5.      else:
6.          print “miss”
7.          if C is full:
8.              _ , evict_block = extract_max(C)       O(log K)
9.              print “evict”
10.     add (future[i], sequence[i]) to C              O(log K)
11.     Inverse[sequence[i]] = futures[i]
```

自行設計測資評估執行時間，並分析時間與空間複雜度

實測兩種方法:

A -> 上述提及Cache使用紅黑樹

B -> Cache每次尋找furthest-in-future時遍歷整個cache(使用array)複雜度O(NK)

平均時間(s)	N = 1e6, K=100	N = 1e6, K=1000	N = 1e6, K=10000	理論時間複雜度	理論空間複雜度
A	1.736	1.760	1.623	O(NlogK)	O(N+K)
B	1.8163	1.750	0.813	O(NK)	O(N+K)

後來根據觀察發現測資生成方式不夠隨機，有cache很大時有大量hit出現，我實作紅黑樹的作法反而會較慢

自行設計測資評估執行時間，並分析時間與空間複雜度

- 後來改了一下隨機的算法，實驗結果如下:

平均時間(s)	N = 1e6, K=100	N = 1e6, K=1000	N = 1e6, K=10000	理論時間複雜度	理論空間複雜度
A	3.226	3.295	3.436	O(NlogK)	O(N+K)
B	3.828	21.679	227.723	O(NK)	O(N+K)

這次實驗就能看出使用紅黑樹實作cache明顯較優秀