**Cover page**

Lee Doe (ldXXXX)

COSC-XXXX

Mock Project - Cache Replacement Policies

**Status of assignment:** Complete

**Time spent on project:** 15 hours

**Things you wish you had been told prior to being given the assignment:** I was a little confused at first about implementing caching when there is no value being retrieved. I should have asked my TA sooner to clear this up.

**Design Document / README**

Language: python (3.6)

I only used standard library functions, except for the following external libraries (which I got permission to use from the TAs):

numpy (1.17.0) -- for random replacement baseline

pandas (0.24.2) -- for writing csv output file in exp\_additional.py

How to run:

python file [--size size]

Example:

python case\_1.txt --size 15

You can also use "python exp\_additional.py" to reproduce all results I have in my report (although there may be differences in the random baseline).

I used a class hierarchy to make it easy to swap out different cache policies for experiments. For each policy, I use dict and list objects as the storage mechanism. For the FIFO and LRU cache, information about the first item added or least recently used item (respectively) are stored in a \_priority list. This could be improved for higher-efficiently implementations with a deque data structure to improve insertion performance at the head of the list, but I just used a list here because it was simpler and has no effect on the cache miss rate. For LFU, I used a dict to store key-value pairs of the item and its frequency. This is efficient for updating the frequency information, but slow to find the item with the lowest frequency. I could have perhaps used a min-heap here, but again it has no effect on the miss rate so I implemented it using the easier (dict) approach. The Random Replacement baseline simply uses a numpy hidden state to select a random item to expire.

**Report & Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | LRU | FIFO | LFU | RR (baseline) |
| Case 1 | 1.000 | 1.000 | 1.000 | **0.600** |
| Case 2 | 0.385 | 0.390 | **0.335** | 0.390 |
| Case 3 | 0.245 | 0.300 | **0.200** | 0.305 |

**Table 1: Miss rate of each policy with cache size = 10.** The best result for each policy is listed in bold. I added the Random Replacement (RR) policy as a baseline.

I report the miss rate of each of the cache policies in Table 1, as well as the Random Replacement (RR) policy as a baseline. The cache strategies exceeded the RR baseline performance in Cases 2 and 3. When I examined the Case 1 file, I noticed that it simply repeatedly iterates sequentially through 15 samples in the same order each time. This means that unless the cache has enough size to hold all items (in this case 15), cache policies that rely on how recently or frequently a value was used (LRU, LFU, and FIFO) will always miss (yielding the observed miss rate of 1.0). In practice, with a predictable access pattern such as this one, it may be better to batch, parallelize, or pipeline the data access of subsequent requests. Or, if these are not practical, to never expire items from the cache (for instance, always keep the first 10 items retrieved, regardless of what other data is observed). This is the reason why the RR baseline does better in this case; there is a probability that a given item is not expired from the cache prior to being re-requested.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Item** | **Case 1** | **Case 2** | **Case 3** |  | **Item** | **Case 1** | **Case 2** | **Case 3** |
| A | 14 | 8 | 65 |  | N | 13 | 12 | 2 |
| B | 14 | 22 | 37 |  | O | 13 | 7 | 2 |
| C | 14 | 14 | 12 |  | P | 0 | 0 | 3 |
| D | 14 | 12 | 17 |  | Q | 0 | 0 | 2 |
| E | 14 | 17 | 10 |  | R | 0 | 0 | 1 |
| F | 13 | 20 | 14 |  | S | 0 | 0 | 1 |
| G | 13 | 8 | 13 |  | T | 0 | 0 | 1 |
| H | 13 | 13 | 2 |  | U | 0 | 0 | 1 |
| I | 13 | 10 | 2 |  | V | 0 | 0 | 1 |
| J | 13 | 10 | 2 |  | W | 0 | 0 | 1 |
| K | 13 | 22 | 2 |  | X | 0 | 0 | 2 |
| L | 13 | 11 | 2 |  | Y | 0 | 0 | 1 |
| M | 13 | 14 | 2 |  | Z | 0 | 0 | 2 |

**Table 2: Item frequencies by case.** Case 1 is largely uniform, whereas Cases 2 and 3 are skewed.

Although the RR baseline strategy appears beneficial in Case 1, it is detrimental in Cases 2 and 3. As shown in Table 2, the frequencies by which each item appears are skewed in these cases, whereas it is largely uniform in Case 1. Furthermore, Case 2 makes use of only 15 items (like Case 1), whereas Case 3 uses 26. By examining the order in which the items appear in Cases 2 and 3, it seems mostly random. Although Case 3 has more distinct items, only 7 are particularly frequent, with the remaining 19 letters only having a frequency of 1 or 2. Thus, I would expect the caching strategies to work better on these, because they can cache these very frequent items. Indeed, all the policies have a lower miss rate for Case 3, despite having more distinct items. Of the four policies, LFU seems to be most effective. This also makes sense because it picks up on the fact that the first 7 items are very frequent. Similarly, LRU performs well because an item that is accessed frequently (given a random occurrence probability) will also likely be used recently. FIFO and the RR baseline perform worse because they are more likely to expire one of the frequently-occurring items in this case.





**Figure 1: Miss rates by cache size.**

To gain a better idea of the effect of the cache size, I decided to run each of the policies up to a cache size of 30. (This is beyond the maximum number of distinct items: 26). I observe that for Case 1, the LRU, FIFO, and LFU policies all have a miss rate of 1 until there is enough space to fit all the items in the cache (15). The RR baseline method does not fall into this trap for the reasons listed above. For Cases 2 and 3, LRU, FIFO, and LFU exhibit similar shapes. However, LFU appears to reach its plateau sooner for Case 3 than the other methods. In fact, it can reach a comparable miss rate to FIFO at 10 cache size with only a cache size of 5. The RR baseline exhibits funky behavior. For instance, in Cases 2 and 3, the miss rate can actually increase by adding cache spots. This is due to the randomness introduced by the policy, and is clearly an undesirable quality of the policy.

In conclusion, this project demonstrated that it is important to consider the types of cache requests that will be made when choosing a cache policy. From the experiments conducted, LFU seems to be a solid choice, unless the requests are made in a very uniform and predictable way. Although the RR baseline may look appealing on the surface for this case, it comes with some serious drawbacks, and probably should not be used in practice.