Assignment 2: Probabilistic Filters

Randomized Algorithms

March 2018

In this assignment, you are going to work with probabilistic filters. A probabilistic filter is a space and time efficient data structure that solves a problem of a set membership identification. Such filters have a one-sided probability of error. If we consider the problem of membership test, the negative answer (not a member) provides us with full confidence of the result, whereas the opposite response (is a member) is affected by the error.

Further, we are providing an overview of two such probabilistic data structures: Bloom filter and Cuckoo filter.

1 Bloom filter

1.1 Overview

Bloom filter was originally conceived by Burton Howard Bloom in 1970[1]. In the simplest implementation, the element can be added to the filter, but not removed. Some modifications support counting, but we are going to leave them out of consideration for now.

Necessary components of a Bloom filter are the storage f of size N and a set of k hash transforms, which return values in the range $1..N$. The values generated by these hash transforms are called *keys*, and they are used as indices of array f. Insertion procedure is given by the algorithm 1.

Algorithm 1 Bloom Filter: Insertion

```
1: $x \leftarrow$ value to insert
2: $f \leftarrow$ Bloom filter
3: for $h_i \in h$ do
4:     $key = h_i(x)\%f.size$
5:     $f[key] = 1$
6: end for
```

The insertion procedure consists of applying hash transforms to the input value to get k keys and using these keys to set the corresponding cells of Bloom filter to 1.

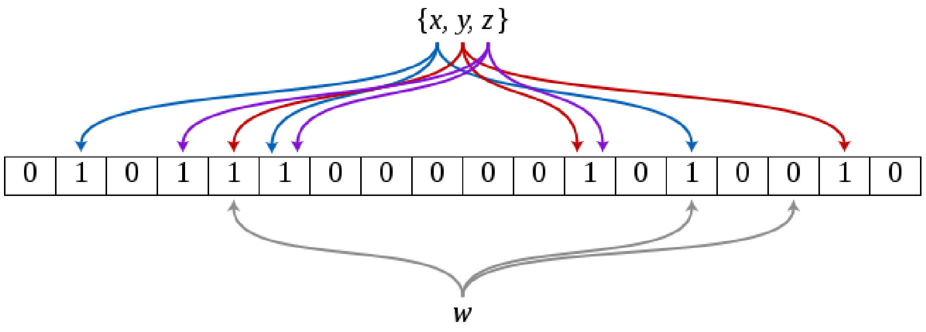


Figure 1: An example of inserting three elements $\{x, y, z\}$ in the Bloom filter of size $m = 18$. Arrows point to locations in the Bloom filter addressed by each of $k = 3$ hash transforms. The element w was not added to the filter before, and one of its hash transforms points to an empty filter cell. (Fig. is the courtesy of Wikipedia)

Membership testing requires similar steps, except that we check the state of filter cells instead of setting them. We can be sure that the item of interest is not a member of a filter whenever any cell of the Bloom filter referred by any of keys is not set. The number of hash transforms k specifies the upper bound for time complexity of membership testing.

Algorithm 2 Bloom Filter: Membership Test

```
1: $x \leftarrow$ value to check
2: $f \leftarrow$ Bloom filter
3: for $h_i \in h$ do
4:     $key = h_i(x) \% f.size$
5:     if $f[key] == 0$ then
6:         return False
7:     end if
8: end for
9: return True
```

1.2 Probability of Error

Since hash transforms are pseudo-random, the probability of a particular filter cell being addressed by a hash transform is $1/N$. Therefore the probability of a particular cell not being updated is $1 - 1/N$. If we assume that the keys are randomly distributed then the probability of a specific element not being addressed after all k keys have been hashed is $(1 - 1/N)^k$.

The error occurs when the membership test returns *True* when the element was not actually in the filter. This can happen only when all keys returned by hash transforms are addressing only the cells that were previously set to 1. The analytic probability that an element being tested is hashed to already altered cells by all k hash transforms is FPP:

$$FPP = (1 - (1 - 1/N)^{mk})^k \tag{1}$$

where m is the number of elements currently in the filter.

The equation above governs the probability of False Positives and is based on the assumption that the hash transforms are independent. Based on this equation one can obtain a lower bound on space complexity, and the number of hash transforms that minimize the probability of False Positives (FPP).

1.3 Choosing Parameters

Conventional specification for Bloom filter includes the membership test complexity and space requirement. Both of these depend on the number of elements we plan to store in the filter. The space complexity for Bloom filter is usually measured in bits per entry and can be calculated from the equation (2).

$$BPE = \lceil log_2(e) \cdot log_2\left(\frac{1}{FPP}\right)\rceil \tag{2}$$

This equation defines the minimal possible space requirement for a single element that allows achieving the desired probability of error. The number of bits here includes both: bits that are set to 1, and bits that remain 0. The space requirement for the filter can be calculated as $BPE \cdot m$, where m is the number of elements in filter.

Filter size has the direct relationship with the number of elements stored in it. Thus, adding a new extra element to the filter will deteriorate its performance with respect to false positives. To avoid this from happening, we need to create a new filter from scratch or provide for extra space at the moment of designing requirements. Thus, we introduce an additional parameter, *working load*, that specifies the percentage of utilized filter cells. Setting *working load* to less than 1.0 will allow us to add elements to the filter in the future without violating FPP requirement. Proper rounding should be applied on each of the steps of these calculations. The total number of elements can be found as

$$elements = \lceil\frac{requirement}{workingload}\rceil$$

The number of hash transforms that minimizes FPP for a given size of Bloom filter (N cells) and the number of expected elements m, is given in (3).

$$k = \lceil ln(2) \cdot \frac{N}{m}\rceil \tag{3}$$

It is worth noting that the value of k returned by (3) is optimal with respect to FPP for given m and N. In some situations, the optimal value of k ensures FPP much less than we require on the expense of testing time complexity. Henceforth, additional tweaking may be required to pick minimal values of N and k that satisfy FPP requirement.

2 Cuckoo Filter

2.1 Overview

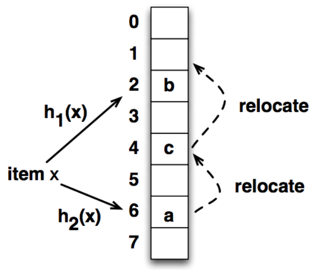
Cuckoo filter is another probabilistic data structure that, just as Bloom filter, supports fast and space efficient membership tests. Cuckoo filter was introduced in 2014 by *Fan et al.* in a paper with title *Cuckoo Filter: Practically Better Than Bloom* [2].

The idea of this filter is based on Cuckoo hash table. Essential components are N buckets; each contains K cells. Unlike Bloom filter that stores binary values in its cells, Cuckoo filter stores fingerprints of inserted elements. Each fingerprint has the length of f_b bits. The size of the fingerprint plays a crucial role in determining the probability of False Positives.

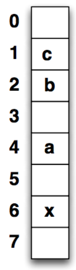
Algorithm 3 Cuckoo Filter: Insertion

```
1: $x \leftarrow$ value to insert
2: $N \leftarrow$ number of buckets
3: $f = fingerprint(x)$
4: $i_1 = hash(x)\%N$
5: $i_2 = (i_1 \oplus hash(f))\%N$
6: if bucket $i_1$ or bucket $i_2$ has empty cell then
7:     add $f$ to that cell
8:     return Done
9: end if
   // must relocate existing item $k$ = randomly pick $i1$ or $i_2$
10: for $n = 0; n < MaxNumKicks; n{+}{+}$ do
11:     randomly select entry $e$ from bucket $k$
12:     swap $f$ and $e$
13:     $k = k \oplus hash(f)$
14:     if bucket k has empty cell then
15:         add $f$ to bucket $k$
16:         return Done
17:     end if
18: end for// filter considered full
19: return Fail
```

When inserting a new value, its compact representation is calculated using *fingerprint* transformation. Each item can be assigned to one of two possible buckets, which are determined by keys i_1 and i_2. If the buckets are full, one of the elements of a randomly chosen bucket is reassigned to another one. The details of insert and test procedures are described in algorithms 3 and 4 correspondingly.



(a) before inserting item *x*



(b) after item *x* inserted

Figure 2: Inserting a value into a Cuckoo filter with number of elements per bucket equal to 1 [2]

Algorithm 4 Cuckoo Filter: Membership Test

```
1: $x \leftarrow$ value to check
2: $f = fingerprint(x)$
3: $i_1 = hash(x)$
4: $i_2 = i_1 \oplus hash(f)$
5: if bucket $i_1$ or $i_2$ has $f$ then
6:     return True
7: end if
8: return False
```

2.2 Probability of Error

Cuckoo filter is notable for the behavior of its probability of error when the utilization of the available space is close to 100%. FPP for this filter is approximated with the equation (4)

$$FPP = 1 - (1 - 1/2^{f_b})^{2 \cdot b \cdot l} \tag{4}$$

where f_b is the size of the fingerprint in bits, b is the bucket size (number of cells per bucket, and l is the filter utilization (added elements/capacity).

2.3 Choosing Parameters

Equation 5 determines the number of bits per entry (BPE) that will allow achieving the desired probability of error

$$BPE = \lceil log_2\left(\frac{1}{FPP}\right) + log_2\left(2b\right)\rceil \tag{5}$$

Naturally, we would want to take the upper limit when rounding this value. To calculate the space required for storing the filter, we need to specify the desired capacity. In this case, besides providing the expected number of elements in the filter, it is useful to also define the *working load* parameter that determines the number of additionally available cells.

3 Task

Disclaimer: data structures that focus on the performance should be implemented using appropriate tools. In this assignment you are going to work with Python, which in general is not suited to solve these kind of problems, but provides good interpretability and ease of managing your code.

You are provided with the template where Bloom and Cuckoo filter are partially implemented. To complete this assignment, you need to finish the following tasks:

1. (8 points) Complete the implementation of *BloomFilter* constructor.
2. (8 points) Complete the implementation of the method *add* for *BloomFilter*. It should support adding a single element and a list of elements.
3. (8 points) Complete the implementation of *__contains__* method for *BloomFilter* (the input is a single element).
4. (8 points) Complete the implementation of *current_fpp* method for *BloomFilter* that computes FPP based on the number of inserted elements (BloomFilter.added is the counter).
5. (8 points) Complete the implementation of *CuckooFilter* constructor.

6. (8 points) Complete the implementation of the method *add* for *CuckooFilter*. It should support adding a single element and a list of elements.

7. (8 points) Complete the implementation of *__contains__* method for *CuckooFilter* (the input is a single element).

8. (8 points) Complete the implementation of *current_fpp* method for *CuckooFilter* that computes FPP based on the number of inserted elements (CuckooFilter.added is the counter).

9. (9 points) The file *run.py* implements a testing procedure, where FPP is tested for different numbers of elements in filters. Obtain a graph that shows theoretical and empirical dependencies of FPP from number of added elements for *BloomFilter* and *CuckooFilter*.

10. (9 points) If you implemented everything correctly, the empirical performance of Cuckoo filter should be worse then the theoretical one. Pay attention to how the fingerprint for the inserted value is calculated (the method *fingerprint*). Should the value of fingerprint have any prohibited states? Fix the issue and produce a new graph.

11. (9 points) Analyze the dependency of space occupied by Bloom and Cuckoo filter from the required probability of false positives.

12. (9 points) Using the formulas (2) and (3) you have calculated the space requirements and the number of hash transforms for Bloom filter. Describe an algorithm that would find the minimal value of filter size N for a given value of test complexity k. (Do not use brute-force search)

References

[1] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.