0.1 Bloom filters

- *Algorithm:* add (algo. 1), query (algo. 2)
- *Input:* an element e to add or query
- *Complexity:* $\mathcal{O}(k)$
- *Data structure compatibility:* An m-bit array A with k hash functions
- *Common applications:* cache filtering, data synchronization, chemical/bio-informatics structure searching, etc.

Problem. Bloom filters

Bloom filter is a type of data structure intended for fast and space-efficient element query. A simple Bloom filter is implemented with an m-bit array and a series of hash functions, the number of which denoted by k. However, it is possible to produce false positive results (where the filter returns true even though the element doesn't exist), thus the structure parameters should be carefully decided.

Description

Bloom filter is a data structure proposed by Burton Howard Bloom in 1970 [1], which is intended for a fast judgement whether an element exists in a set. It requires a constant time and space to maintain a set of elements. On the other hand, it is a probabilistic data structure, which means that the search result might not be definite. It would yield *definitely not existing* or *probably existing* answers, i.e. there would be false positive results [6]. Another point is that the element can only be added but not be deleted [6].

To set up a Bloom filter, we need a binary array A with m bits, all initially set to be 0. We denote the i-th bit as A_i. Besides, k different hash functions are needed, each mapping an element e onto the m positions in A. The hash functions should be non-correlated and their results should be uniformly distributed.

To add1 an element e to the set, feed it to the hash functions, and set the array bits corresponding to the hash results to be 1.

To query2 whether the element e exists in the set, feed it to the hash functions as well. If any one of the array bits corresponding to the hash results is 0, then this element is definitely not existing. On the other hand, if all of the corresponding bits are 1, this element probably exists with a false positive rate ϵ.

Algorithm 1: Add new element,htbp

```
Input   : new element $e$
1 for $i \leftarrow 1$ to $k$ do
2     $index \leftarrow Hash_i(e)$
3     set $A_{index} = 1$
4 end for
```

The number of hash functions k is decided by the element number n and array length m. For given m, n, the value k that minimizes the false positive rate is given by

$$k = \frac{m}{n} \ln 2,$$

and it gives the minimum false positive rate

$$\epsilon \approx \left(1 - e^{-kn/m}\right)^k \to \ln \epsilon = -\frac{m}{n}(\ln 2)^2$$

Algorithm 2: Query,htbp

```
Input   : element to query $e$
Output: whether $e$ exists
1 for $i \leftarrow 1$ to $k$ do
2     $index \leftarrow Hash_i(e)$
3     if $A_{index} = 0$ then
4         return False
5     end if
6 end for
7 return True
```

It also suffices to show the optimal *bits per element* m/n is approximately $-1.44 \log_2 \epsilon$. Here gives a deduction for the above equations [3].

Bloom filter has unusual time complexity properties. We may easily see that no matter for adding or querying, what we need to do is always computing the k hash values and looking up the array, which means the time complexity is simply $\mathcal{O}(k)$.

The space of a Bloom filter is constant $\mathcal{O}(m)$, since Bloom filter does not need to really store the elements to search them like conventional deterministic data structures, but contains hash values as characteristics instead. However due to the possibility of false positive results, Bloom filter does not always have advantages over deterministic data structures. In case of a limited potential range of elements, a traditional deterministic data structure is sufficient, while Bloom filter is preferred when the range size is large or hard to estimate [4].

As discussed above, the allowing of false positive leads to saving in space and runtime, hence both simple Bloom filters and their extensions can be applied in cases where we only need to maintain the existence of set elements. One typical database applications is cache filtering. According to an analysis, over 3/4 of the URLs requested by net users are "one-hit-wonders", which would be accessed only once and never again [5]. It would be a waste if there are cached on disks. A Bloom filter can identify whether a URL has been requested before and cache it if so, filtering the "one-hit" urls and saving plenty of disk space. Andrei and Michael's survey also mentioned a "more widespread attention in the networking literature" for Bloom filters [2]. They concludes four main types of jobs where overlapping is also possible, which are collaborating in overlay and peer-to-peer networks, resource routing, pack routing and measurement [2]. Further applications include bio-informatics field, where Bloom filters are used to generate space-efficient biometric rotation-invariant feature representations [7].

References.

[1] Burton H Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: *Communications of the ACM* 13.7 (1970), pp. 422–426 (cit. on p. 1).

[2] Andrei Broder and Michael Mitzenmacher. “Network applications of bloom filters: A survey”. In: *Internet mathematics* 1.4 (2004), pp. 485–509 (cit. on p. 2).

[3] Atul Kumar. *Bloom Filters – Introduction and Python Implementation.* URL: https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/ (cit. on p. 2).

[4] llimllib. *Bloom Filters by Example.* URL: https://llimllib.github.io/bloomfilter-tutorial/ (cit. on p. 2).

[5] Bruce M Maggs and Ramesh K Sitaraman. “Algorithmic nuggets in content delivery”. In: *ACM SIGCOMM Computer Communication Review* 45.3 (2015), pp. 52–66 (cit. on p. 2).

[6] Stefan Nilsson. *Bloom filters explained.* URL: https://yourbasic.org/algorithms/bloom-filter/ (cit. on p. 1).

[7] Christian Rathgeb, Frank Breitinger, Christoph Busch, and Harald Baier. “On application of bloom filters to iris biometrics”. In: *IET Biometrics* 3.4 (2014), pp. 207–218 (cit. on p. 2).