Empirical Analysis of Bloom Filters

Barnabas Forgo

Abstract

A Bloom filter is a popular data structure that solves the set membership problem in a probabilistic manner. This report will examine the inner workings and time complexity of two versions of this data structure via a reference implementation. It will be shown how some mathematical properties of the Bloom filter can be used to optimise its reliability. Finally, these properties are confirmed through empirical evidence, using a dataset that is similar to real-world use cases.

Introduction

Testing whether an element is part of a set is a very common problem in computer science with many existing solutions. One such solution is a so called *Bloom filter*, created by Burton Howard Bloom in 1970 [1]. This data structure uses hash functions to create a space-efficient 'fingerprint' of the set, that can be queried in constant time. The trade-off to achieve this compact representation of the set is that it allows for *false positives* when querying, but not for *false negatives*, that is, it returns "possibly in set" or "definitely not in set". Because of this uncertainty, this data structure is considered *probabilistic*.

Example use cases

Such a data structure has many potential use cases. The original paper introducing the algorithm [1] detailed a use case of automated hyphenation software. This example is based on the assumption that 90% of words can be hyphenated using simple rules and the remaining 10% requires consulting a lookup table to get the correct hyphenation. Since at that time it was not feasible to store this table in memory it had to be delegated to a disk, making the lookup quite expensive. Bloom's experiments resulted in reducing unnecessary disk access by 84% over a dictionary of 50 000 words and only using ~37KB memory.

A more modern example of Bloom filters was implemented in Google Chrome [2], [3]. This browser uses a Bloom filter of approximately 1 million URLs to stop the user from visiting a malicious site. When the filter reports that a site might be malicious the URL is checked against a larger online database, to confirm that it is not a false positive.

A somewhat less conventional implementation was created by Akamai (a large-scale content delivery network) to prevent "one-hit wonders", i.e. files that are only accessed once, from entering a disk cache [4]. To achieve this, they used the content hash of the files to populate a Bloom filter, and they would only let them to be stored in the cache if the file was already in the filter. Using this method, they reduced disk writes by up to 50%, and as a side effect it also decreased read latency and increased cache hit rate.

Operations

There are two key ingredients to a Bloom filter:

1. An array of bits.
2. A number of hash functions that map elements to locations in the array.

The size of the array (m) and the exact number of hash functions (k) to be used have to be selected when the Bloom filter is created. Since these parameters affect the performance, and reliability of the filter they have to be chosen carefully. If the number of elements in the set is known ahead of time then the two parameters can be calculated to achieve any desired false positive probability.

Any hash function can be used to create a Bloom filter. When choosing a specific hashing algorithm there are two factors to pay attention to. The quality of the hash function affects the actual number of false positives; a good hash maps uniformly across the array. The performance of the Bloom filter is dominated by the number of hashes (as shown later), therefore, choosing a faster algorithm results in a faster filter.

Insert

For example, let us take a Bloom filter of programming languages to show how the two operations work in detail. Let us set the parameters to $m = 16$ bits and $k = 3$ hash functions and insert "JavaScript". To do this, first the value has to be hashed by each of the three functions: [33, -18, 8]. To ensure that the hashed values are within the bounds of the bit array so that they can be used as indices, two operations are applied on each value, modulo by m and absolute: [1, 2, 8]. Finally, each location indicated by the transformed hash values are set to 1 in the array. This process is visualised in Figure 1 below.

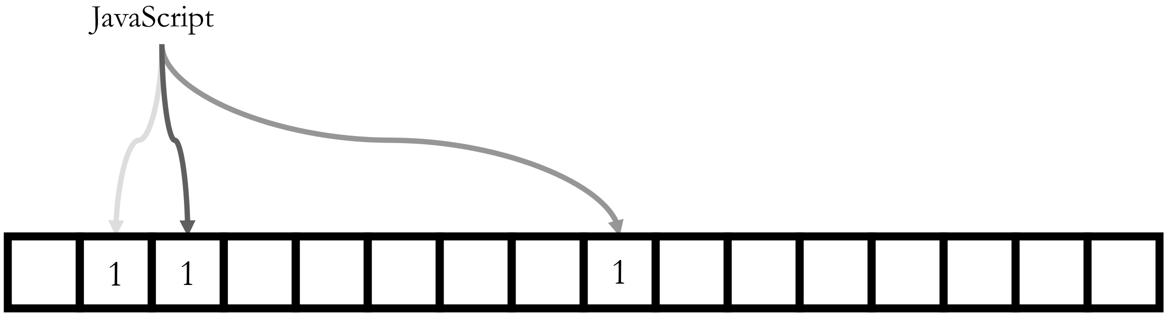

Figure 1 Inserting JavaScript

To insert additional elements the same process is repeated. When another two elements are inserted ("Haskell" and "Rust"), a hash collision can be observed, where two different elements map to the same bit in the array. It is easy to see from Figure 2 below, that for this phenomenon a delete operation is not possible on the classic Bloom filter, because there is no way of knowing whether a bit maps to one or more elements. There is an alternative version of the data structure called the "counting Bloom filter" [5] that does support a delete operation but it consumes ~4 times more memory.

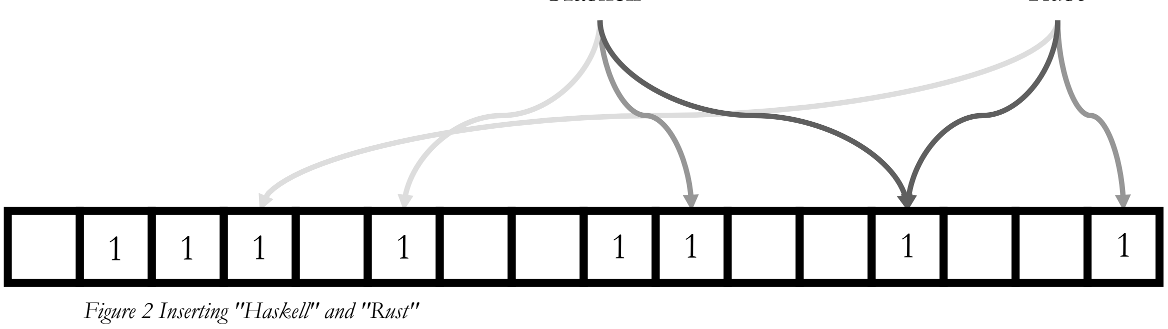

Figure 2 Inserting "Haskell" and "Rust"

In Figure 3 below a reference implementation of insert can be seen in TypeScript [6]. Let us go through the implementation line by line and analyse its time complexity. Starting from `computeHashes`, each hash function goes through the same pipeline: evaluation against the current value, modulo by m, and finally absolute. Each of the operations in this pipeline run in constant time ($O(1)$) as none of them depend on the number of elements in the filter. However, it is important to note that evaluating the hash function runs usually in linear time with respect to the size of the element being hashed. This is why it is important to pick a fast hashing method.

Since the pipeline has to be evaluated for every hash function therefore `computeHashes` has a runtime complexity of $O(k)$. Returning to `insert`, there is only one operation left to do: setting the bits in the array for each hash. Assuming that arrays can be indexed in constant time, it can be said that `insert` runs in $O(k)$ as well.

```
private computeHashes(value: T): number[] {
  return this.hashFunctions
    .map(fn => fn(value))
    .map(newHash => newHash % this.bits.length)
    .map(Math.abs);
}

insert(value: T) {
  this.computeHashes(value)
    .forEach(hash => this.bits[hash] = 1);
}
```

Figure 3 Implementation of insert in TypeScript

Membership Test

To query a Bloom filter whether it has an element, the queried element has to be hashed the same way as in `insert`, by evaluating each hash function over the element and then ensuring that the values conform to the bounds of the bit array. By definition of a hash function one element always maps to the same locations in the array, therefore if all of the locations pointed by the hashes are one, then the element is part of the set. Unless, there are multiple hash collisions and by accident an element hashes to locations that are already ones from other elements; this is called a *false positive*. Let us see all the possible outcomes of query visualised in Figure 4 below:

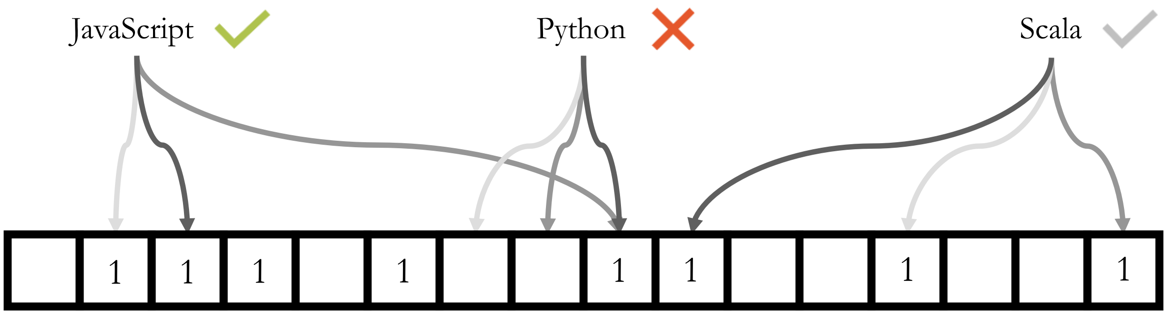

Figure 4 Membership test over three elements, and three different outcomes

- ☐ "JavaScript" – *positive*: Since this element was added previously (cf. Figure 1), therefore the hashes will point to locations that were set to ones.
- ☐ "Python" – *negative*: This element was not added to the filter and some of the hashes point to bits that are zeros, indicating that this element is not part of the set.
- ☐ "Scala" – *false positive*: Even though this element was not added to the filter, due to multiple hash collisions it still evaluates as positive.

Logically, there should be another outcome, a *false negative*, however a Bloom filter does not allow for this, by design. Since there are no operations that flip ones to zeros (making ones permanent), therefore a false negative is not possible.

To analyse the runtime complexity of this operation let us examine a possible implementation in TypeScript on the side in Figure 5. This operation uses the same `computeHashes` method, as earlier (cf. Figure 3), which is known to have a runtime complexity of $O(k)$. After the k hashes are computed, each is mapped to the bit it points to, finally, the bits are checked whether they are set using a reducer. Since these operations run in constant time, and each have to be executed k times it can be said that query runs in $O(k)$ as well.

```
private computeHashes(value: T): number[] {
  return this.hashFunctions
    .map(fn => fn(value))
    .map(newHash => newHash % this.bits.length)
    .map(Math.abs);
}

query(value: T): boolean {
  return this.computeHashes(value)
    .map(hash => this.bits[hash])
    .reduce(
      (result, bit) => result && bit,
      true
    );
}
```

Figure 5 Implementation of query in TypeScript

Kirsch-Mitzenmacher Optimisation

After analysing both operations available on a Bloom filter, it is easy to see that the main bottleneck in performance is the computation of hash values, as this is the only non-trivial operation. If this part of the algorithm can be optimised, then it would lead to a significant performance gain.

Adam Kirsch and Michael Mitzenmacher achieved exactly this in 2006 [7], by proving both mathematically and empirically, that using a common technique in hashing literature called *double hashing* in a Bloom filter does not increase the asymptotic false positive probability. Double hashing means that arbitrary number of hash functions can be simulated by only evaluating two hashes and then recombining them using the following formula: $g_i(x) = h_1(x) + i\, h_2(x)$.

To implement this optimisation in the TypeScript example only the `computeHashes` method has to be changed as shown in Figure 6. Again, any hashing algorithm can be used to compute the hash values, MD5 and CRC is only an example here.

After this optimisation, the asymptotic runtime of both `insert` and `query` are still $O(k)$, but with a much smaller constant factor as the hash functions do not have to go through the elements over and over again.

```
private computeHashes(value: T): number[] {
  const hash1: number = md5(value);
  const hash2: number = crc(value);
  return range(0, this.hashFunctions)
    .map(index => hash1 + index * hash2)
    .map(newHash => newHash % this.bitmap.length)
    .map(Math.abs);
}
```

Figure 6 Implementing the optimisation in TypeScript

Runtime Complexity Experiment

To prove that the operations in fact run in $O(k)$ the following experiment was devised. A Java [8] implementation was created that can switch between the standard and optimised version of the data structure and time a number of insertions precisely. The data used to conduct the experiments was a set of known malicious and benign URLs compiled from two sources [9], [10]. This dataset seemed appropriate as it reflects a real-world problem, akin to the Chrome example presented earlier.

Many deterministic set membership data structures have a runtime complexity that is dependent on the number of elements inserted (e.g. TreeSet worst case search: $O(\log n)$, HashSet worst case search: $O(n)$). It would be beneficial to confirm that Bloom filters' complexity does not depend on this parameter. For this reason, the time taken to insert n elements into a filter with k hash functions was measured. In theory, executing `insert` n times should run in $O(nk)$, and this can be clearly observed in Figure 7.

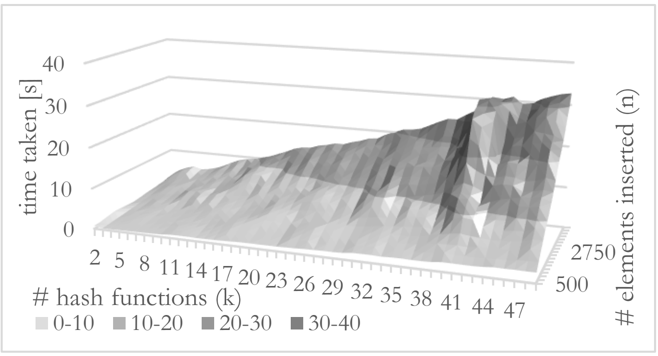

Figure 7 Inserting n elements into a bloom filter with k hash functions

The optimised version of the data structure performed the same way, however 3-4 times faster than the classic version for most experiments, as expected. To compare the two results the time taken for inserting 5000 elements into a filter with k hash functions was plotted in Figure 8. It is interesting to note that the classic algorithm outperforms the newer one when only a few hash functions are used, which is expected, as at least two hash values are computed per element even when only one is required.

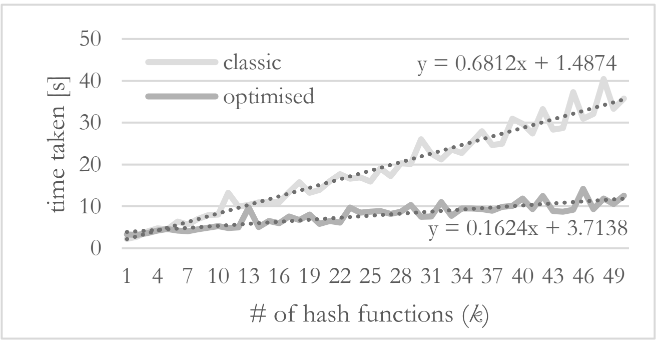

Figure 8 Inserting 5000 elements into a filter with k hash functions

Note that these experiments only include tests over `insert`. Plotting the results for `query` is unnecessary as they look identical to the ones for insert. This is due to the fact that both operations are dominated by the shared hashing step.

FALSE POSITIVE PROBABILITY EXPERIMENT

Theorem: The false positive probability of a Bloom filter can be estimated using

$$p \approx \left(1 - e^{-k/b}\right)^k$$

where
k is the number of hash functions,
n is the number of element in the set,
m is the size of the array, and
$b = m/n$, or the number of bits per element.

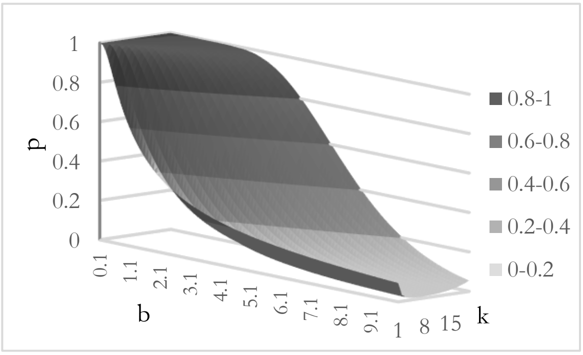

Figure 9 Expected false positive probability for certain b and k

Proof:

This is assuming that the filter is built using perfect hash functions that are independent and distribute elements uniformly. The probability of a bit remaining zero after setting a single bit: $1 - 1/m$. After one insertion that requires setting k bits: $(1 - 1/m)^k$. After inserting the whole set: $(1 - 1/m)^{kn}$. The probability of a bit being set to one: $1 - (1 - 1/m)^{kn}$. Since $1 + x \approx e^x$ for small x, therefore this can be simplified as $1 - e^{-kn/m} = 1 - e^{-k/b}$ (using $x = -1/m$). The probability that k different bits are set to one: $\left(1 - e^{-k/b}\right)^k$. **QED**.

Unfortunately, there are no (general) perfect hash functions in the real-world, but let us see if this assumption holds up with imperfect hashing. To do this an experiment was created, that measures the false positive probability of certain b bitrate and k hash functions, so that it can be compared with the expected false positive probability.

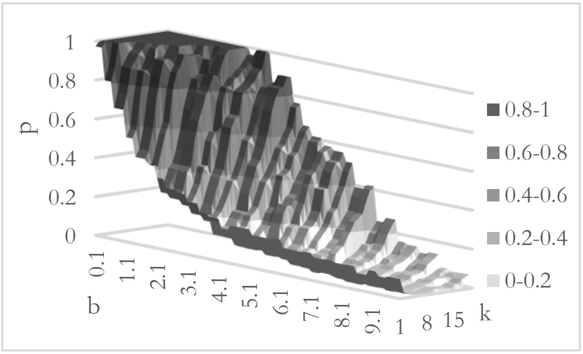

Figure 10 Actual false positive probability

Comparing the expectations to the actual results in Figure 9 and Figure 10, it is clear that despite some outliers there is a very strong correlation between the two datasets ($r = 0.996$). This means that the above equation can be used to estimate the false positive probability of real filters. The plot shown above was generated using the classic Bloom filter, over the URL dataset. The optimised version has the same behaviour, albeit with slightly more false positives.

Using this mathematical property of a Bloom filter, formulae can be devised that can calculate the optimal number of hash functions ($k_{optimal}$) and the optimal number of bits per element ($b_{optimal}$), given any desired false positive probability ($p_{desired}$).

$$b_{optimal} = -\frac{log_2\, p_{desired}}{ln\, 2} \approx -1.4\, log_2\, p_{desired}$$

$$k_{optimal} = b_{optimal}\, ln\, 2 \approx 0.7 b_{optimal}$$

Let us examine whether the above formula can be used to correctly estimate the optimal number of hash functions. To do this the data from the previous experiment was used. For each b the minimal and predicted minimal k was compared. From the differences between the two values the Pareto chart in Figure 11 was created. It is clear that this estimation of the required k is fairly accurate as it finds the optimal k about 55% of the time and it is within 1% error more than 80% of the time.

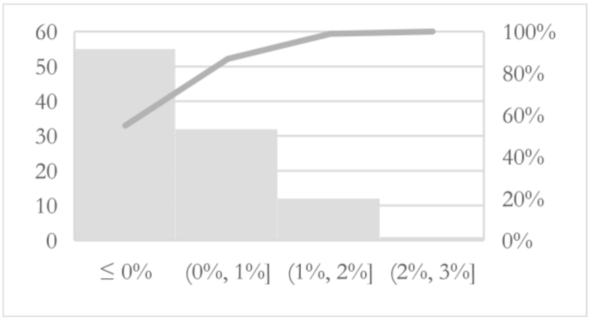

Figure 11 Predicted vs. actual minimal false positive rates

Limitations & Alternatives

One of the key shortcomings of a Bloom filter is the fact that it is probabilistic in nature. If false positives are unacceptable in the problem domain than a Bloom filter cannot be used. In this case it might be worthwhile to use a hash table or a tree based set implementation. However, it is important to note that most of the deterministic set data structures do require storing the elements. If the set is too large to keep in memory then a probabilistic data structure is the only way.

There are different versions of Bloom filters that optimise one of the two resources (space & time consumption) but at the cost of the other [11]. Alternatively, a *cuckoo filter* [12] can be used that outperforms Bloom filters in most aspects at the cost of a slightly slower insertion speed and possible rejection of inserted elements (at high loads). Another substitute that can be used to solve similar problems is *hash compaction* [13] (a compacted version of hash tables) that offers the highest accuracy among these structures at the cost of linear space complexity.

Conclusion

To summarize, it has been shown how a bit array and some hash functions can be used to probabilistically encode sets in a space efficient manner. Using a reference implementation, it was proven that the operations, `insert` and `query`, on this data structure both run in $O(k)$ time, which is most importantly, independent of the number of elements in the set. Using a dataset that is similar to real-world usages of the data structure, it was confirmed empirically that this analysis is correct. Furthermore, it was shown through experiments, how some mathematical properties of this algorithm can be exploited to optimise its false positive probability.

References

[1] B. H. Bloom, ‘Space/time trade-offs in hash coding with allowable errors’, *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[2] Google, ‘Chrome Web Browser’. [Online]. Available: https://www.google.com/chrome/. [Accessed: 31-Dec-2017].

[3] A. Yakunin, ‘Nice Bloom filter application’, *Alex Yakunin’s blog*, 2010. [Online]. Available: http://blog.alexyakunin.com/2010/03/nice-bloom-filter-application.html. [Accessed: 31-Dec-2017].

[4] B. M. Maggs and R. K. Sitaraman, ‘Algorithmic nuggets in content delivery’, *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 3, pp. 52–66, 2015.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, ‘Summary cache: a scalable wide-area Web cache sharing protocol’, *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[6] Microsoft, ‘TypeScript - JavaScript that scales.’ [Online]. Available: https://www.typescriptlang.org/. [Accessed: 08-Jan-2018].

[7] A. Kirsch and M. Mitzenmacher, ‘Less hashing, same performance: building a better bloom filter’, in *ESA*, 2006, vol. 6, pp. 456–467.

[8] Oracle, ‘java.com: Java + You’. [Online]. Available: https://java.com/en/. [Accessed: 09-Jan-2018].

[9] PhishTank, ‘PhishTank > Developer Information’. [Online]. Available: https://www.phishtank.com/developer_info.php. [Accessed: 09-Jan-2018].

[10] F. Ahmad, ‘Using Machine Learning to Detect Malicious URLs’, *KDnuggets*, 2016. [Online]. Available: https://www.kdnuggets.com/2016/10/machine-learning-detect-malicious-urls.html. [Accessed: 09-Jan-2018].

[11] F. Putze, P. Sanders, and J. Singler, ‘Cache-, Hash- and Space-Efficient Bloom Filters’, in *Experimental Algorithms*, 2007, pp. 108–121.

[12] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, ‘Cuckoo filter: Practically better than bloom’, in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[13] U. Stern and D. L. Dill, ‘A new scheme for memory-efficient probabilistic verification’, in *Formal Description Techniques IX*, Springer, 1996, pp. 333–348.