Notes Accompanying Today's Class in Algorithm Design

Roberto Grossi
Università di Pisa

Feb. 28, 2020

1 Bloom Filters, Cuckoo Hashing, and Succinct Rank Data Structure

These notes are based on [3, 4]. Consider a set S of n keys chosen from a universe U.

(1) For a given (1-side) error probability $0 < f < 1$, we learned that Bloom filters achieve probability f using $k \approx (m/n) \ln 2$ hash functions that map $U \to [m]$. They take $O(k)$ time, and use nearly $(\log(1/f)/\ln 2)n \approx 1.44 \log(1/f)n$ bits of space.

(2) We learned that Cuckoo hashing, using two hash functions $h_1, h_2 : U \to [m]$, achieves worst-case constant-time lookup, by checking at most two positions indicated by these hash functions.

(3) Today, we look at a succinct Rank data structure R, which takes as input a bitvector B of m bits, where n of them are 1s. The constant-time supported operation is $rank_B(j)$ which returns the number of 1s in the first j bits of B. Space is $\lceil \log \binom{m}{n} \rceil + o(m)$ bits for the entire structures (no need to store B explicitly[1]), where $\lceil \log \binom{m}{n} \rceil$ is the information-theoretic lower bound for storing a binary string of length m with n 1s (equivalently, a set of n elements from a universe size m) [2, 1].

We show that using the data structures (2) and (3) we can improve the bounds of Bloom filters in 1) when S is static (i.e. S does not change over time) and $\log(1/f)$ is a power of two.

Specifically, we see how to obtain a 1-side error probability f for lookup/membership using nearly $\log(6/f)n$ bits: as $\log(6/f) \approx 2.58 + \log(1/f)$, we have an additive constant instead of a multiplicative in the space bound for (1), which is much better (e.g. try with $f = 10^{-6}$). Moreover we use just three hash functions and lookup takes constant time.

Fingerprints. The first idea is to choose, randomly and uniformly, a hash function $h \in \mathcal{H}$ from a universal hash family $\mathcal{H}$ (as the one seen in class), where $h : U \to [m']$.

We thus define $S' = \{h(x) \mid x \in S\}$, where $|S'| \leq |S| = n$. When we want to test, given any $y \in U$, whether $y \in S$, we lookup $h(y) \in S'$. What is the lookup error? If $y \notin S$ but $h(y) \in S'$, we have that there exists $x \in S$ such that $h(y) = h(x)$. And we saw that the latter collision probability is one over the range of the hash function, namley, $\Pr_{h \in \mathcal{H}}\{h(y) = h(x)\} = 1/m'$. Thus the probability that there exists $x \in S$ such that $h(y) = h(x)$ is n/m' by the union bound on x. In order to get the same error f as the Bloom filter in (1), we need to fix $f = n/m'$.

[1]We observe that $\log \binom{m}{n} \leq m$, thus R is always preferred instead of storing B explicitly.

Given $x \in S$, note that $h(x)$ uses $\log m' = \log(n/f)$ bits and is called its signature. The elements of S' require $\log(n/f)\, n$ bits in total: we only store S', not S to save space as each key in S could be very large (same motivation as Bloom filters). But still space is too much.

In the following, we want to store S' in little additional space and access in constant-time.

Cuckoo hashing. Cuckoo hashing uses two randomly and independently chosen hash functions $h_1, h_2 \in \mathcal{H}$, where $h_1, h_2 : [m'] \to [m]$ and $m = 3|S'| \leq 3n$.[2] Lookup to check whether $y' \in S'$ takes constant time as it probes locations $h_1(y')$ and $h_2(y')$ in a table T of m entries.

Are we happy? We use fingerprint h, but this time we fix $m' = 2/f$. Given any $y \in U$, we check whether $y \in S$ by computing its fingerprint $y' = h(y)$ and checking whether $y' \in S'$ in constant time. We observe that now the probability that there exists $x \in S$ such that $h(y) = h(x)$, is $2/m' = f$ by the union bound on x, as x can stay in just two positions.

But what about the space? Since T uses $m \leq 3n$ entries, each capable of storing $\log(2/f)$ bits, we use a total of $3 \log(2/f)\, n$ bits, more than twice those required by the Bloom filters in (1)!

We observe that we waste space for at least $2n$ empty entries of T. To put a remedy on that we proceed as follows.

- We mark with a 1 which positions in T contains a nonempty entry, and 0 othwerise. This yields a bitvector B of m bits, where $|S'| \leq n$ of them are 1s. In the following, let us assume $|S'| = n$ wlog. Recall that $m = 3n$.
- We pack the n nonempty entries of T into an array P of n entries. Note that P stores a permuation of the elements in S', and thus takes $\log(2/f)\, n$ bits.

We observe that the nonempty entries in T in left-to-right order are in 1-to-1 correspondence with the 1s in B and the elements in P, both in left-to-right order. Thus the ith nonempty entry in T corresponds to the ith 1 in B and the ith element in P.

Now, in order to check whether $y' \in S'$ in constant time using cuckoo hashing, we need to check whether $T[h_1(y')] = y'$ or $T[h_2(y')] = y'$. Since we do not want to use T anymore, we equivalently perform the following test.

1. If $B[h_1(y')] = B[h_2(y')] = 0$, then $y' \notin S'$ (and thus $y \notin S$, with no error).
2. Otherwise, let $B[h_1(y')] = 1$, wlog. If $B[h_1(y')]$ is the ith bit 1 in B, we test whether $P[i] = y'$. Same test when $B[h_2(y')] = 1$.

Note that the missing piece in the puzzle is how to test if $B[h_1(y')] = 1$ is the ith bit 1 in B. Letting $j = h_1(y')$, this requires to check whether $B[j] = 1$ (easy), and there are i 1s in the first j bits of B. For the latter, we need to introduce and use the Rank succinct data structure in (3).

Rank data structure. The input is a bitvector B of m bits, where n of them are 1s. We want to replace B with a succinct Rank data structure R that answser constant-time $rank_B(\cdot)$ queries. Recall that $rank_B(j)$ returns the number of 1s in the first j bits of B. Note that $B[j] = 1$ iff $rank_B(j) \neq rank_B(j-1)$, so it is enough to store R in place of B.

The best implementations of R use $\lceil \log \binom{m}{n} \rceil + o(m)$ bits. Thus we can replace T in cuckoo hashing with P and R. Hence, we can simulate Bloom filters with our claimed bounds, storing three hash functions h, h_1, h_2, which take $O(\log(1/f) + \log n)$ bits, plus P, which takes $\log(2/f)\, n$

[2] In class we saw that $m > 2cn$ for any constant $c > 2$, but the choice $m = 3n$ works fine as we saw.

bits, plus R, which takes $\lceil \log \binom{m}{n} \rceil + o(m) \approx n \log(m/n) + o(m) = n \log 3 + 0(n)$ bits as $m = 3n$. Overall this is $\log(6/f)\, n + o(n)$ bits as claimed.

In the class, we described a less space-efficient implementation of R for illustrative purposes. It uses $3m + o(m)$ bits, but it gives an idea on how R works.

Let $\ell = (1/2) \log m$. We build, using the so-called Four-Russians trick, a two dimensional table L of $2^\ell \times \ell = O(\sqrt{m} \log m)$ entries. Entry $L[\alpha, j'']$ returns the number of 1s contained in the first j'' bits of binary string α. We build L by brute force, generating all binary strings α of length ℓ, and scanning each of them for each j''. Since there are $2^\ell = O(\sqrt{m})$ such strings α, we take $O(\sqrt{m}\, \mathrm{polylog}(m)) = o(m)$ time to build it. Moreover, since each entry of L uses $O(\log \log m)$ bits, the space occupied by L is $O(\sqrt{m}\, \mathrm{polylog}(m)) = o(m)$ bits. Clearly, L can be queried in constant time.

Now, consider B and partition it into chunks of ℓ bits each. Each chunk is a string α, so we can use L to compute in constant time how many 1s are found in the first j'' bits of α. Because of that, we can conceputally see B as an array B' of m/ℓ chunks. We store an array C, so that $C[t]$ explicitly contains an integer that tells how many 1s are found in the the first $t-1$ chunks of B'. Array C uses $m/\ell \cdot \log m = 2m$ bits. Hence, L, B, and C occupy a total of $3m + o(m)$ bits to implement R.

In order to answer $rank_B(j)$, let us take the chunk of B within which j falls. It corresponds to $\alpha = B'[j']$, where $j' = 1 + \lfloor j/\ell \rfloor$. Observe that the jth bit in B is the j''th bit in α where $j'' = 1 + j \bmod \ell$. Thus we return $C[j'] + L[\alpha, j'']$ as the value of $rank_B(j)$, in constant time.

Simpler Approach. Suppose that we stick to the first choice of the fingerprtins, where $m' = n/f$. Let S' be stored in this way. Using the Rank data structure in (3), we can store S' in place of S. Note that while the universe of S is U, now the universe of S' is $[n/f]$. Exercise: Show that we get almost $\log(1/f)n + o(n/f)$ bits in this way.

Lower bound. Let us see why $\log(1/f)\, n$ bits are needed to store S with membership error f. This is of general interest in all applications where sets can be stored approximately to save space (e.g. the inverted lists in search engines when estimating the number of occurrences of searched terms).

In order to give a full picture, let U be the universe from which the keys of S are taken, and let $F \subseteq U \setminus S$ be set of false positives.[3] As false positives occur with probability f, we observe that $|F| \leq f|U|$, so we can assume wlog that $|F| = f|U|$.

Suppose that b is the minumum number of bits to store S approximately, so that we can establish membership on S with error f of false positives. That is, the b bits encode sets S and F in our terminolgy, as we provide a positive answer to membership for the elements in $S \cup F$, rather than for the elements of S alone. Note that we can mark the false positives in F, using not less than $\log \binom{|S|+|F|}{|S|}$ bits. In other words, using $b + \log \binom{|S|+|F|}{|S|}$ bits we can store exactly S, without any errors.

By the information-theoretic lower bound, we cannot take less than $\log \binom{|U|}{|S|}$ bits: this translates into the necessary condition $b + \log \binom{|S|+|F|}{|S|} \geq \log \binom{|U|}{|S|}$, from which we get

$$b \geq \log \binom{|U|}{|S|} - \log \binom{|S|+|F|}{|S|}.$$

Using the approximation that $\log \binom{a}{b} \approx b \log(a/b)$ and $\log(1+x) \approx \log x$ for large x, we get that

[3] We do not need to specify here how F is obtained as it is a lower bound. In our setting, when using $h \in \mathcal{H}$, we have that $F = \{y \in U \setminus S \mid \text{ there exists } x \in S \text{ such that } h(x) = h(y)\}$.

$\log\binom{|U|}{|S|} - \log\binom{|S|+|F|}{|S|} \approx |S|\log\frac{|U|}{|S|} - \log\frac{|S|+|F|}{|S|} \approx |S|\log\frac{|U|}{|S|} - \log\frac{|F|}{|S|} = |S|\log\frac{|U|}{|F|} = |S|\log\frac{|U|}{f|U|} = n\log(1/f)$ ignoring lower order terms (here $n = |S|$).

References

[1] M. Patrascu. Succincter. In IEEE, editor, *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science: October 25–23, 2008, Philadelphia, Pennsylvania, USA*, pages 305–313. IEEE Computer Society Press, 2008.

[2] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007.

[3] I. Razenshteyn. Cuckoo hashing for sketching sets. `http://blog.ilyaraz.org/?go=all/cuckoo-hashing-for-sketching-sets/`, 2019. [Online; accessed 28-Feb-2020].

[4] R. Venturini. Simple lower bound for approximate set query. Personal communication, 2020.