CSCI 5454 Spring 2023: Assignment #5
Due Date: Tuesday, March 14

Topics: Randomized Algorithms: Hashtables and Randomized Algorithms.

P1: A *Bloom filter* is a very important data structure. Suppose we see a long stream of M elements:

$$e_1, e_2, e_3, \ldots, e_M$$

Note that the elements can be URLs or requests for accesses to various sectors on a disk. We wish to design a data structure that given an element f, tells us if we have seen f in the stream of elements. Solving this exactly requires us to store all the unique elements in the stream and somehow compare f against it.

If there are n unique elements in the stream and each item has b bits, we will need to store at least nb bits.

A Bloom filter provides an approximate approach to this problem. A Bloom filter is hash table T with m slots where each slot simply holds a true/false value. Thus the overall space usage is simply m bits.

We have k hash functions $h_1, \ldots, h_k$ each of which is assumed to be uniform for the stream data. I.e, the probability for any stream data item e, hash function h_j and slot k of the table ($k \in \{0, \ldots, m-1\}$).

$$\mathbb{P}(h_j(e) = k) = \frac{1}{m}$$

Initially all slots in the table are set to false.
Insertion: Inserting an element e – set the slots $T[h_1(e)], T[h_2(e)], \ldots, T[h_k(e)]$ all to true.

Query: If we wish to know whether an element f was already seen, we simply check the slots $T[h_1(f)], T[h_2(f)], \ldots, T[h_k(f)]$. If they are all set to true, then we conclude that f *may* have been seen before. Otherwise, if any of them is false, we conclude that f was definitely not seen.

Note that a Bloom filter can suffer from the problem of *false positives*. It could claim that an element f was present in the stream even if f were not. This happens because the slots $T[h_1(f)], T[h_2(f)], \ldots, T[h_k(f)]$ were all set to true but by insertion of different stream elements, potentially. However, if the Bloom filter concludes that an element f is not present in a stream, then it is not present.
(A, 2 points) Think of a setting where we have m bins and we are dropping balls into bins where each ball can land uniformly into any of the m bins, independent of where the previously dropped balls landed. Suppose we drop nk such balls into m bins. What is the probability that bin number j is unoccupied (i.e, it has zero balls land in it)?
(B, 3 points) Assume that the probability that a particular bin is occupied is independent of the probability that a different bin is occupied. Show that the probability that some set of k bins $\{j_1, \ldots, j_k\}$ are all occupied is $\leq (1 - e^{-nk/m})^k$.

Note: we assume mutual independence of the events that bin i and bin j are occupied for two different bins i, j in problem (B).

(C, 10 points) Suppose we throw some number q balls uniformly into $m > 2$ bins. What is the precise probability that bins 1 and bins 2 are both occupied at the end? Do not assume independence

for this part. (Use the equality: $P(E_1 \cap E_2) = P(E_1) + P(E_2) - P(E_1 \cup E_2)$, where E_1 and E_2 denote the events that that bins 1 and 2 are occupied respectively.

Note: Using part (C), we conclude that the probability that a ball lands on one bin is *not* independent of the probability that it lands on another. Thus, our solution in part (B) is not appropriate for estimating the false positive rate of a Bloom filter. However, in practice, it is a fairly good approximation. In fact, for the previous problem, notice that you may assume independence if it so happened that $(1 - 2/m)^q \approx (1 - 1/m)^{2q}$.

(D, 5 points) Let us fix $k = 10$ hash functions and assume that the balls/bins model for Bloom filter is true. Assume that the false probability is bounded by $(1 - e^{-nk/m})^k$. Let $m = 8 \times 10^6$ bits (roughly 1 MB).

How many unique elements can we insert while keeping the false positive rate bounded by 0.01? If each element is a URL that takes on the average 40 bytes to store, how much space have we saved using a Bloom filter?

P2: We studied the randomized min cut algorithm in class. Consider the following version:

- Compress the graph from n to $n/4$ vertices (instead of $n/2$ vertices).
- Call the algorithm recursively four times on the $n/4$ vertices and take the best answer.

Here we compress the graph from n to $n/4$ nodes by contracting $3n/4$ edges for each recursive call on a graph of size n.
(A, 5 point) What is the probability that we do not contract any of the min-cut edges during the initial step of compressing the graph from n to $n/4$ edges? Derive an exact expression in terms of n and approximate it by a constant assuming large enough n.
(B, 5 point) Suppose performed $k = 4$ repetitions of the algorithm over the contracted graphs of size $n/4$, what is the overall running time?
(C, 5 point) Show that the success probability for a single repetition of the recursive algorithm is $\Omega(1/n)$. Follow the same trick that was explained in class and the notes posted online.
Note: Using parts (A)-(C), we conclude that we would need $Cn\ln(n)$ repetitions of the modified algorithm to achieve success with overall probability of at least $1-1/n^c$ for some constant c, yielding an overall complexity of $\Theta(n^3 \ln(n))$. This is clearly less optimal than the $\Theta(n^2(\ln(n))^3)$ complexity we obtained in class.

P3: Consider the following modification of the algorithm instead:

- Compress the graph from n to $n/2$ vertices.
- Call the algorithm recursively $k = 2$ times on the $n/2$ vertices and take the best answer (instead of $k = 4$ times).

Suppose we performed $k = 2$ instead of 4 repetitions of the recursive algorithm.
(A, 5 points) What is the overall running time?
(B, 5 points) Solve for the probability of success by setting up a new recurrence relation and using the same trick as in class/notes.

In fact, we conclude that using 2 instead of 4 repetitions yields a very similar outcome as the previous problem.