Karlsruhe Institute of Technology

Stefan Walzer, Stefan Hermann
ITI Prof. Sanders
winter term 2025/2026

Exercise Sheet 8 – Bounded Differences and Bloom Filters

Probability and Computing

Exercise 1 – Balls, Bins and Bounded Differences

Let $\lambda > 0$ be a constant, $m = \lambda n$ the number of balls, and n the number of bins. The j-th ball is placed in bin X_j, where $X_1, \dots, X_m \sim \mathcal{U}([n])$ are independent random variables. The collision count is defined as $C = |\{(i,j) \mid 1 \le i < j \le m, X_i = X_j\}|$. The goal of this exercise is to derive a concentration bound for C.

(i) Show that $\mathbb{E}[C] = \Theta(n)$.

For $i \in [n]$, let $L_i = |\{j \in [m] \mid X_j = i\}|$ denote the load of bin i. It is neither difficult nor interesting to show that $\Pr[\max_{i\in[n]} L_i \le \log n] \ge 1 - O(n^{-100})$. Assume this to hold in the following (proof provided in the sample solution).

(ii) Define $C_{\text{cap}} := \sum_{i\in[n]} \binom{\min(\log n, L_i)}{2}$. Show that $\Pr[C_{\text{cap}} = C] = 1 - O(n^{-100})$.

(iii) Show that $\Pr[C_{\text{cap}} - \mathbb{E}[C_{\text{cap}}] \ge t] \le \exp(-2t^2/(m \cdot \log^2 n))$.

(iv) Show that $\Pr[C - \mathbb{E}[C] \ge n^{2/3}] = O(n^{-100})$.

(v) Assume an algorithm inserts n keys into a hash table and then queries every key exactly once. Show that the algorithm runs in time $O(n)$ except with probability $O(n^{-100})$ under suitable assumptions. Note that we are *not* talking about *expected* running time.

Solution 1

(i) Any two distinct balls collide with probability $1/n$. Hence, $\mathbb{E}[C] = \binom{m}{2} \cdot \frac{1}{n} = \frac{m(m-1)}{2n} = \frac{\lambda(m-1)}{2} = \Theta(n)$.

We now examine the claimed bound on bin loads. Fix arbitrary $i \in [n]$ and $k \in \mathbb{N}$. For a subset $S \subseteq [m]$ of size k, let $E_{S,i}$ denote the event that all balls in S are placed in bin i. Then $\Pr[E_{S,i}] = n^{-k}$. It follows that:

$$\Pr[\max_{i\in[n]} L_i \ge k] = \Pr\left[\bigcup_{\substack{S\subseteq[m],i\in[n]\\|S|=k}} E_{S,i}\right] \overset{\text{UB}}{\le} \sum_{\substack{S\subseteq[m],i\in[n]\\|S|=k}} \Pr[E_{S,i}] = \binom{m}{k} \cdot n \cdot n^{-k} \le \frac{m^k}{k!} n^{-k+1} = \frac{\lambda^k n}{k!}.$$

Substituting $k = \log n$ and observing that $\lambda^k = \text{poly}(n)$ while $(\log n)! = n^{\Omega(\log \log n)}$ grows faster than any polynomial, completes the argument.

(ii) If $\max_{i \in [n]} L_i \leq \log n$, then

$$C_{\text{cap}} = \sum_{i \in [n]} \binom{L_i}{2}.$$

The latter is an alternative definition of the collision count C (summing collisions within each bin).

(iii) C_{cap} is a function of the m independent random variables $X_1, \ldots, X_m$. Changing any single X_j alters C_{cap} by at most $\log n$. The result follows directly from McDiarmid's inequality.
Remark: The same strategy does not apply well to C itself. In the extreme case where $X_1 = \cdots = X_{m-1} = 1$ and $X_m = 2$, changing X_m to 1 induces $m - 1$ additional collisions.

(iv) We combine the previous two results:

$$\begin{aligned}
\Pr[C - \mathbb{E}[C] \geq n^{2/3}] &\leq \Pr[C \neq C_{\text{cap}} \vee C_{\text{cap}} - \mathbb{E}[C] \geq n^{2/3}] \\
&\overset{\text{UB}}{\leq} \Pr[C \neq C_{\text{cap}}] + \Pr[C_{\text{cap}} - \mathbb{E}[C] \geq n^{2/3}] \\
&\overset{C \geq C_{\text{cap}}}{\leq} \Pr[C \neq C_{\text{cap}}] + \Pr[C_{\text{cap}} - \mathbb{E}[C_{\text{cap}}] \geq n^{2/3}] \\
&\leq O(n^{-100}) + \exp(-2n^{4/3}/(m \log^2 n)) = O(n^{-100}).
\end{aligned}$$

(v) We use hashing with linear chaining at load factor $\alpha = 1$ and make the simple uniform hashing assumption. The distribution of keys to buckets matches the setting at hand with $\lambda = 1$. Inserting the n keys takes $O(n)$ times and querying the keys takes $O(n + C)$ time because very collision between two keys $x \neq y$ increases either the query time of x or the query time of y by one step. From (i) we know that $\mathbb{E}[C] = O(n)$ and from (iv) we know that $\Pr[C \geq 2\mathbb{E}[C]] \leq O(n^{-100})$. This implies a running time of $O(n + 2\mathbb{E}[C]) = O(n)$, except with probability $O(n^{-100})$.

The following exercise is not directly related to randomized algorithms, except that we used the bound in lecture. Hence, "Bonus".

Exercise 2 – Bonus: Approximations of e

You know that $e = \lim_{n \to \infty} (1 + \frac{1}{n})^n$ (this is even a common *definition* of e). Derive from this that the following two inequalities hold for all $n \in \mathbb{N}$:

$$\begin{aligned}
(1 + \tfrac{1}{n})^n \leq e \leq (1 + \tfrac{1}{n})^{n+1} \\
(1 - \tfrac{1}{n})^n \leq e^{-1} \leq (1 - \tfrac{1}{n})^{n-1}.
\end{aligned}$$

The following steps are suggested:

(i) Prove the left-hand inequalities.
Hint: Use again $1 + x \leq e^x$.

(ii) Show that the right-hand sides are monotonically decreasing in n.

(iii) Deduce the right-hand inequalities from (ii) and a limit argument.

Solution 2

(i) We have:

$$(1 + \tfrac{1}{n})^n \leq (e^{1/n})^n = e$$
$$(1 - \tfrac{1}{n})^n \leq (e^{-1/n})^n = e^{-1}$$

(ii) First, taking logarithms of the hint for (i), we obtain:

$$\forall x \in (-1, \infty) : \ln(1 + x) \leq x$$

To show that $f(n) = (1 + \frac{1}{n})^{n+1}$ is monotonically decreasing in $n \in \mathbb{N}$, it suffices to show that $\ln(f(x)) = (x + 1)\ln(1 + \frac{1}{x})$ is monotonically decreasing in $x \in (0, \infty)$. Consider its derivative and show it is everywhere ≤ 0:

$$(\ln(f(x)))' = \ln(1 + \tfrac{1}{x}) + \frac{x+1}{1+\frac{1}{x}} \cdot (-\tfrac{1}{x^2}) \leq \tfrac{1}{x} - \frac{x+1}{(x+1)x} = 0.$$

Analogously, for $g(x) = (1 - \frac{1}{n})^{n-1}$, we show the derivative of $\ln(g(x))$ is ≤ 0:

$$(\ln(g(x)))' = ((n-1)\ln(1 - \tfrac{1}{n}))' = \ln(1 - \tfrac{1}{x}) + \frac{x-1}{1-\frac{1}{x}} \cdot \frac{1}{x^2} \leq -\tfrac{1}{x} + \frac{x-1}{(x-1)x} = 0.$$

(iii) We have by separating a factor:

$$\lim_{n\to\infty} (1 + \tfrac{1}{n})^{n+1} = \lim_{n\to\infty} (1 + \tfrac{1}{n})^n \cdot \lim_{n\to\infty} (1 + \tfrac{1}{n}) = e \cdot 1 = e.$$

The limit of $(1 - \frac{1}{n})^{n-1}$ can be computed as follows:

$$\lim_{n\to\infty} (1 - \tfrac{1}{n})^{n-1} = \lim_{n\to\infty} (\tfrac{n-1}{n})^{n-1} = \lim_{n\to\infty} \left(\frac{1}{\frac{n}{n-1}}\right)^{n-1} = \frac{1}{\lim_{n\to\infty} (\frac{n}{n-1})^{n-1}}$$
$$= \frac{1}{\lim_{n\to\infty} (1 + \frac{1}{n-1})^{n-1}} = \frac{1}{e} = e^{-1}.$$

Thus, the right-hand sides converge to e and e^{-1}, respectively. By (ii), they converge "from above". Hence, the inequalities hold as claimed.

Exercise 3 – Counting Bloom Filter (a.k.a.: Count-Min Sketch)

A Counting Bloom Filter is initially like a standard Bloom Filter: it manages n keys in an array of size m, with load factor $\alpha := \frac{n}{m}$ and k hash functions. The parameters are chosen as in lecture so that a standard Bloom Filter would have false-positive probability ε. The array now contains natural numbers instead of bits. In addition to insert and query, a delete operation is now available.

Algorithm insert(x):
  for $i \in [k]$ **do**
    $A[h_i(x)] + +$

Algorithm delete(x):
  for $i \in [k]$ **do**
    $A[h_i(x)] - -$

Algorithm query(x):
  for $i \in [k]$ **do**
    if $A[h_i(x)] = 0$ **then**
      return false
  return true

We allow a key to be inserted *multiple times* into the Counting Bloom Filter. Hence, the managed object S is now a *multiset* with n elements $\{x_1, \ldots, x_n\}$, each with multiplicity $a_1, \ldots, a_n$. The operation insert(x) adds one copy of x to the multiset S, i.e., increments the multiplicity of x if x is already in S, or adds a new element with multiplicity 1 if x is not yet in S. The meaning of delete is analogous. As before, query may return false positives but not false negatives.

(a) We require that delete is called only for elements that are actually in S (with multiplicity at least 1). What breaks if we do not enforce this?

Additional useful operations can be implemented with Counting Bloom Filters.

(b) Implement an operation count that, for $x \in D$, returns an estimate count(x) of the multiplicity a of x in S. Show that $\Pr[a \neq \text{count}(x)] \leq \varepsilon$.

(c) The primary argument for using (Counting) Bloom Filters is their low memory footprint compared to exact data structures. We should therefore avoid using large integer types (e.g., 64 bits) for counters. Consider an application where "most" counters never exceed 8 bits. We therefore use an array A of 8-bit counters. Briefly discuss the following proposals for handling counter overflows. What are the advantages and what compromises are made?

- Alice merely prevents counter overflows, i.e., adapts the $A[h_i(x)] + +$ operation in insert and the $A[h_i(x)] - -$ operation in delete so that the counter is incremented/decremented only if the maximum/minimum representable value has not yet been reached.
- Bob proposes to "freeze" a counter that reaches $(11111111)_2 = 255$, i.e., no future insert or delete operations will modify this counter.
- Carol proposes to use the bit string $(11111111)_2 = 255$ as a marker indicating that the true counter value exceeds 254. In this special case, the true counter value is stored in a hash table.

Solution 3

(a) A delete(y) for a y that was never inserted reduces k counters in the Counting Bloom Filter uncontrollably. Thus, counters may become 0. Consequently, for some $x \in S$, query(x) might return false, yielding a false negative — which is not allowed.

(b) We return the minimum of the counters associated with x:

Algorithm count(x)**:**
- $r \leftarrow \infty$
- **for** $i \in [k]$ **do**
  - $r \leftarrow \min(r, A[h_i(x)])$
- **return** r

Note that count(x) $< a$ is impossible, since every occurrence of x is counted by every associated counter. However, it is possible that count(x) $> a$ if all k counters are also incremented by other keys. To understand this, let $A'[1..m] \in \{0,1\}^m$ be the standard Bloom Filter into which all elements of S except x have been inserted (using the same hash functions as for the Counting Bloom Filter). Then:

$$\begin{aligned}
\text{count}(x) \neq a &\Leftrightarrow \text{count}(x) > a \\
&\Leftrightarrow \min_{i \in [k]} A[h_i(x)] > a \\
&\Leftrightarrow \forall i \in [k] : A[h_i(x)] > a \\
&\Leftrightarrow \forall i \in [k] : \text{some key in } S \text{ other than } x \text{ uses } A[h_i(x)] \\
&\Leftrightarrow \forall i \in [k] : A'[h_i(x)] = 1 \\
&\Leftrightarrow x \text{ is a false positive for } A'
\end{aligned}$$

The latter event has probability at most ε by the choice of configuration parameters. Hence, the same holds for the equivalent first event.

(c) Regarding the proposals:

- Alice's proposal is simple to implement, but false negatives become possible. The simplest example is inserting the same key x 256 times and then deleting it 255 times. The final insertion is lost, and the deletions reset all relevant counters to 0. Subsequently, query(x) returns false, although x should still be in the data structure. Not good.
- Bob's proposal ensures counters never falsely return to zero, so false negatives are impossible. However, frozen counters can never be released. In the long run, the false-positive rate may increase.
- Carol's proposal makes no compromises in functionality. The additional hash table naturally consumes space, and accesses to it incur time costs.

Exercise 4 – Estimating Set Intersections with Bloom Filters

Let $n, m, k \in \mathbb{N}$. Alice and Bob wish to estimate how similar their musical tastes are. Let X be Alice's n favorite songs and Y be Bob's n favorite songs. The goal is to estimate $\gamma := \frac{|X \cap Y|}{n} \in [0, 1]$. They proceed as follows:

- Alice constructs a Bloom filter $A[1..m] \in \{0,1\}^m$ for X using k hash functions $h_1, \ldots, h_k$.
- Bob constructs a Bloom filter $B[1..m] \in \{0,1\}^m$ for Y using the *same* k hash functions.
- Alice and Bob exchange their filters and compute $\delta := \frac{|\{i \in [m] | A[i] \neq B[i]\}|}{m}$.
- Alice and Bob compute an estimate $\bar{\gamma}$ for γ based on δ.

Solve the following subproblems:

(a) Discuss: What advantages and disadvantages might this method have compared to direct exchange of X and Y?

(b) Gain intuition: What values of δ do you expect (approximately) for the extreme cases $\gamma = 1$ and $\gamma = 0$?
Hint: You may assume here and in the following that the Bloom filters use an "optimal" configuration with $\alpha k = \ln(2)$.

(c) Compute $\mathbb{E}[\delta]$ as a function of γ. You may drop lower-order terms, e.g., write $(1 - \frac{1}{m})^m \approx e^{-1}$ without carrying an $o(1)$ term.
Hint: At first glance, other parameters (e.g., $n, m, k, \alpha, \varepsilon$) might appear relevant. Their influence vanishes in lower-order terms.

(d) Discuss: Which concentration bound is suitable for proving that δ is close to $\mathbb{E}[\delta]$ with high probability?

(e) Rearrange the equation from (c) to show how an estimate $\bar{\gamma}$ for γ can be computed from δ.

(f) Speculate: What role does the choice of k (or ε) play in this context?

Solution 4

(a)
- Advantage: Space usage may be smaller.
- Disadvantage: We can ensure $\bar{\gamma}$ is close to γ with high probability, but cannot compute γ error-free.
- Advantage: Elements of X and Y are not revealed; i.e., Alice can plausibly deny any $x \in X$ without being contradicted.

(b) For $\gamma = 1$, clearly $A[i] = B[i]$ for all i, so $\delta = 0$. For $\gamma = 0$, the two Bloom filters are independent. By lecture, in a Bloom filter with $\alpha k = \ln(2)$, approximately half the entries are 1 and half are 0. Thus, $\Pr[A[i] \neq B[i]] \approx \Pr_{C,D \sim \mathrm{Ber}(1/2)}[C \neq D] = 1/2$, so we expect $\delta \approx 1/2$.

(c) We write $h(z) := \{h_1(z), \ldots, h_k(z)\}$. Note: Events concerning different keys or hash functions can be separated by independence. Let x_0 be an arbitrary element in X and y_0 an arbitrary element in $Y \setminus X$.

$$\begin{aligned}
\mathbb{E}[\delta] &= \frac{\mathbb{E}[|\{i \in [m] \mid A[i] \neq B[i]\}|]}{m} = \frac{1}{m}\sum_{i=1}^{m} \Pr[A[i] \neq B[i]] = \Pr[A[i_0] \neq B[i_0]] \\
&= \Pr[(A[i_0] = 0 \wedge B[i_0] = 1) \vee (A[i_0] = 1 \wedge B[i_0] = 0)] = 2\Pr[A[i_0] = 0 \wedge B[i_0] = 1] \\
&= 2\Pr[\forall x \in X : i_0 \notin h(x) \wedge \exists y \in Y \setminus X : i_0 \in h(y)] \\
&= 2\Pr[\forall x \in X : i_0 \notin h(x)] \cdot \Pr[\exists y \in Y \setminus X : i_0 \in h(y)] \\
&= 2\Pr[\forall x \in X : i_0 \notin h(x)] \cdot (1 - \Pr[\forall y \in Y \setminus X : i_0 \notin h(y)]) \\
&= 2\Pr[i_0 \notin h(x_0)]^{|X|} \cdot (1 - \Pr[i_0 \notin h(y_0)]^{|Y \setminus X|}) \\
&= 2\Pr[i_0 \neq h_1(x_0)]^{k|X|} \cdot (1 - \Pr[i_0 \neq h_1(y_0)]^{k|Y \setminus X|}) \\
&= 2(1 - \tfrac{1}{m})^{k|X|} \cdot (1 - (1 - \tfrac{1}{m})^{k|Y \setminus X|}) \\
&= 2(1 - \tfrac{1}{m})^{kn} \cdot (1 - (1 - \tfrac{1}{m})^{k(1-\gamma)n}) \\
&= 2(1 - \tfrac{1}{m})^{k\alpha m} \cdot (1 - (1 - \tfrac{1}{m})^{k(1-\gamma)\alpha m}) \\
&\approx 2e^{-k\alpha} \cdot (1 - e^{-k(1-\gamma)\alpha}) \\
&= 2e^{-\ln(2)} \cdot (1 - e^{-(1-\gamma)\ln(2)}) \\
&= 1 - \left(\tfrac{1}{2}\right)^{(1-\gamma)}.
\end{aligned}$$

(d) The method of bounded differences (or McDiarmid's inequality) is suitable here. The $k \cdot |X \cup Y|$ relevant hash values are all independent. Changing any one alters δ by at most $\pm\frac{1}{m}$. The lecture's analysis of the concentration of Z (number of zeros) transfers directly.

(e) From (c), we have $\mathbb{E}[\delta] = 1 - (\frac{1}{2})^{(1-\gamma)}$. We drop the "$\mathbb{E}$" (since we have only δ, not $\mathbb{E}[\delta]$) and replace γ with $\bar{\gamma}$ (since we cannot compute γ exactly but only estimate it). Solving $\delta = 1 - (\frac{1}{2})^{(1-\bar{\gamma})}$ yields: $\bar{\gamma} = 1 - \log_2(1/(1-\delta))$.

(f) The larger k, the stronger the concentration bound and the better the estimate $\bar{\gamma}$. However, there is little reason not to use $k = 1$. It is even conceivable to choose $k \in (0, 1)$, meaning a key is assigned a position with probability k and discarded with probability $1 - k$. These random choices must be made via a hash function known to both Alice and Bob. With $k = \Theta(1/n)$, one could achieve memory usage independent of n. Similar to approximation algorithms, one could introduce relative error and failure probability and compute the required k as a function of these parameters.