CS 388R: Randomized Algorithms, Fall 2021 DATE

Lecture 7: Coin Probability Estimation and Cuckoo Hashing

Prof. Eric Price *Scribe: Ethan Lao, Raymond Hong*

1 Determine the probability of a coin

Problem: Given a coin with probability p of heads, estimate p from flipping the coin

Algorithm 1: Flip coin n times. Suppose you flip the coin n times and define X_i as a random variable where $X_i = 1$ if the ith flip is a heads and 0 otherwise. Define $\hat{p}$ as your estimation of p derived by dividing the number of heads by total flips.

$$\hat{p} = \sum_{i=1}^{n} X_i * \frac{1}{n} \tag{1}$$

$$E[\hat{p}] = p \tag{2}$$

Suppose you want a probability estimate within ϵp with probability of error at most δ

$$Pr[|\hat{p} - p| > \epsilon p] = Pr[|\hat{p} - \mu| > \epsilon \mu] \leq 2e^{-\frac{\epsilon^2 np}{3}} \tag{3}$$

Then solving $2e^{-\frac{\epsilon^2 np}{3}} \leq \delta$ for n, you obtain $n \geq \frac{3}{p\epsilon^2} \log(2/\delta)$

So flip $n = \frac{3}{p\epsilon^2} \log(2/\delta)$ times to obtain the desired estimate

Problem: Cannot determine an appropriate value for n if you know nothing about p

If you know a range of values p could be, just choose the value in the range that requires the most flips

Idea: Suppose you perform $n = \frac{3}{p\epsilon^2} \log(2/\delta)$ coin flips

$$E[\sum_{i=1}^{n} X_i] = n * p = \frac{3}{p\epsilon^2} \log(2/\delta) * p = \frac{3}{\epsilon^2} \log(2/\delta) \tag{4}$$

This value does not depend on p

Algorithm 2: Keep flipping coins until you obtain $\frac{10}{\epsilon^2} \log(2/\delta)$ heads. I.e. stop once $\sum X_i = \frac{10}{\epsilon^2} \log(2/\delta) = k^*$

Then your probability estimate is $\hat{p} = \frac{k^*}{n}$. Define the true probability of heads as $p = \frac{k^*}{n^*}$

How good is this estimate?

$$Pr[\hat{p} > (1+\epsilon)p] = Pr[n < \frac{n^*}{1+\epsilon}] \leq Pr[\sum_{i=1}^{\frac{n^*}{1+\epsilon}} X_i > k^*] \tag{5}$$

Can apply Chernoff bound to last probability

$$E[\sum_{i=1}^{\frac{n^*}{1+\epsilon}} X_i] = \frac{n^*}{1+\epsilon} * p = \frac{k^*}{1+\epsilon} \tag{6}$$

$$Pr[\sum_{i=1}^{\frac{n^*}{1+\epsilon}} X_i > k^*] \leq \frac{\delta}{2} \text{ holds for } \frac{k^*}{1+\epsilon} \geq \frac{3}{\epsilon^2}\log(\frac{4}{\delta}) \tag{7}$$

Note this analysis is just for overestimating p (why $\delta/2$ is used), similar analysis can be done for underestimating

2 Cuckoo Hashing

Suppose you are hasing n items into n bins. We have already covered 2 approaches:

1. Standard Hash Tables: Expected $O(1)$ time per operation. Worst case $O(\frac{\log(n)}{\log\log(n)})$ w.h.p. (Recall the balls and bins analysis on max load)
2. Two Choice Approach (i.e. for each item you pick 2 bins and place the ball in the bin with less items): Expected $O(1)$ time per operation. Worst case is $O(\log\log(n))$ with high probability. (Recall the two choice balls and bin analysis on max load)

Cuckoo Hashing offers:

- Expected $O(1)$ per insertion
- Worst case $O(\log(n))$ for insertion
- Worst case $O(1)$ for table lookup

It does so by allowing us to evict balls, guaranteeing that each bin only contains at most 1 ball. Similar to Two Choices, you check two cells and go to the one with less elements. However, if both cells already have a ball, you randomly evict one of the balls. The evicted ball gets moved to the other cell the hash function points to. If this cell is occupied as well, evict and move it, continuing the process until the assignment is stable.

2.1 Graph Representation of Cuckoo Hashing

We can conceptualize the assignment of items to cells using a directed graph, where the vertices represent the cells of the hash table the edges represent the items being hashed. Since each item hashes to 2 locations, the edge representing it will connect the 2 cells to which it hashes to (ie. make an edge between the 2 vertices that represent the cells to which the item hashes to). The direction of the edge determines which cell the item gets assigned to.

What can go wrong with this approach? If you have a cycle, every vertex is used by the cycle. Therefore, if there is an edge connected to a vertex in the cycle (but not a part of it), the edge must direct away from the cycle to maintain that each has 1 ball at most. If the other end of this edge happens to connect to a vertex in another cycle, this creates a problem known as a "barbell" (two connected cycles) where it is impossible to find valid assignment.

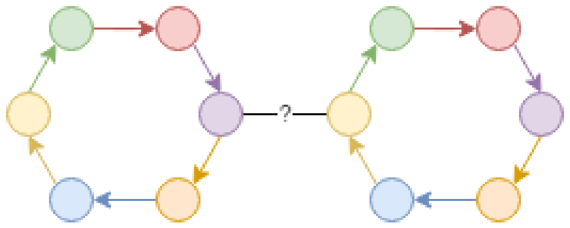

Figure 1: Example of a barbell

2.2 Algorithm and Analysis

Algorithm: Try to insert the item, evicting and moving items as necessary. If you cannot because of a barbell, then pick a new hash function and rebuild the table.

Thus, there are 2 things to show: (I) Rebuilding is a rare occurrence, (II) Insertion is fast on average

2.2.1 Show rebuilding is a rare occurrence

Note that for a barbell to exist, a cycle must exist as well:

$$Pr[\text{barbell exists}] \leq Pr[\text{cycle exists}] \leq \sum_{k=2}^{m} Pr[\text{length } k \text{ cycle exists}] \tag{8}$$

So what is the probability that a length k cycle exists? This is equivalent to the number of possible length k cycles multiplied by the probability that the specific cycles exists. Observe that the number of possible length k cycles is $\binom{n}{k}\frac{k!}{2k}$. This is because there are $\binom{n}{k}$ ways to select vertices and $k!$ ways to order vertices. We divide by $2k$ to compensate for duplicates (a cycle has k starting positions and can run in 2 directions). Thus, we write:

$$
\begin{aligned}
& Pr[\text{length } k \text{ cycle exists}] && (9)\\
&= (\text{number of possible length k cycles}) * (\text{probability specific cycle exists}) && (10)\\
&= \left[\binom{n}{k}\frac{k!}{2k}\right]\left(\frac{m}{\binom{n}{2}}\right)^k && (11)\\
&\leq \left[\left(\frac{en}{k}\right)^k \frac{k^k}{2k}\right]\left[\frac{2m}{n^2}\left(1+\frac{1}{n-1}\right)\right]^k && (12)\\
&\leq \left[(en)^k\frac{1}{2k}\right]\left[\left(\frac{2m}{n^2}\right)^k e\right] && (13)\\
&\leq (en)^k\left(\frac{2m}{n^2}\right)^k e && (14)\\
&\leq \left(\frac{2em}{n}\right)^k e && (15)
\end{aligned}
$$

Now consider the number of cycles for all k:

$$Pr[\text{barbell exists}] \leq \sum_{k=2}^{m} Pr[\text{length } k \text{ cycle exists}] \leq \sum_{k=2}^{n}\left(\frac{2em}{n}\right)^k e \leq 2e\left(\frac{2em}{n}\right)^2 \leq \frac{1}{20} \tag{16}$$

Therefore, the probability of rebuilding is at most $\frac{1}{20}$.

Now let's define $T = \mathbb{E}[\text{time to insert all items and find collisions if one exists}]$. Then:

$$\mathbb{E}[\text{total time}] = T + \frac{1}{20}\,\mathbb{E}[\text{total time}] = \frac{20}{19}T \tag{17}$$

Rebuilding only increases the expected time by a factor of $\frac{20}{19}$.

2.2.2 Show insertion is fast on average

Next, we try to find a better estimate for T, the time to insert all items:

$$T = \sum_{i=1}^{m}\mathbb{E}[\text{time to insert } i\text{th item}] \approx \mathbb{E}[\text{size of component of } i\text{th item}] \tag{18}$$

Claim: for any fixed vertex u, $\mathbb{E}[\text{size of component containing u}] = O(1)$ for $m < \frac{n}{100}$.

We can simplify this claim to examine graphs of $G(n,p)$, where each edge exists with $p = \frac{1}{100}$. This, in turn, can be even further simpliied down to the Galtan-Watson branching process.

In the Galtan-Watson branching process, take an initial vertex and let it have $d = n-1$ potential neighbors. Then, we independently pick edges with probability p. For our case, let's set $p = \frac{1}{100d}$. For each edge picked, let the graph expand to the neighbor vertex, and repeat the process on it's neighbors recursively.

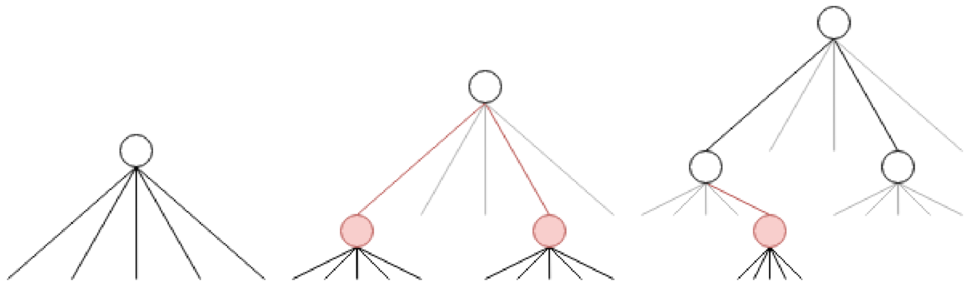

Figure 2: The Galtan-Watson branching process. In this example, the algorithm ends up picking 2 edges in the first layer, then 1 in the next. This process will continue recursively.

If we let $C = \mathbb{E}[\text{number of vertices picked including root}]$, we have:

$$C = 1 + dpC = 1 + \frac{1}{100}C = \frac{100}{99} = O(1) \tag{19}$$

This can be applied to graphs of $G(n,p)$. The key difference is that in $G(n,p)$, we may have an edge between two vertices that we have already visited. However, this will only decrease the component size, so $G(n,p)$ will also have $O(1)$ component size.

Now, we generalize to the original claim by looking at a graph $G\left(n, \frac{2m}{\binom{n}{2}}\right)$. This graph has an expected $2m$ edges, implying that it will have $\geq m$ edges with high probability. Therefore, we can write:

$$\mathbb{E}[\text{Component size in } G(n,m)] \leq \mathbb{E}\left[\text{Component size in } G\left(n, \frac{2m}{\binom{n}{2}}\right)\right] + n * n^{-100} \leq O(1) \tag{20}$$

Note: We add $n * n^{-100}$ because there's very a small probability that $G\left(n, \frac{2m}{\binom{n}{2}}\right)$ had $< m$ edges, but when that happens the component size is always at most n.