CS 388R: Randomized Algorithms, Fall 2019 September 24th, 2019

Lecture 8: Cuckoo Hashing

Prof. Eric Price *Scribe: Maohua Wang, Ruizhe Zhang*

NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

1 Overview

In the last lecture, we solved a few puzzles about balls & bins and random walk.

In this lecture we first introduce the Hash table and Cuckoo Hashing. Then, we prove that Cuckoo Hashing only needs $O(1)$ time per insertion in expectation and $O(1)$ time per lookup in worst-case. We also discuss the "Universality" and k-wise independence of random Hash functions.

2 Hash table

Hash table is a very commonly used data structure which has two operations: Insert(key,value) and lookup(key). In previous lectures, we studied the max-load of balls & bins problem, which is closely related to the collisions of random hash functions. Hence, we summarize the known results as follows:

- One choice Hashing
  - Expected $O(1)$ time per operation[1].
  - Worst case $O\left(\frac{\log n}{\log \log n}\right)$ time per operation.
- Two choices Hashing
  - Expected $O(1)$ time.
  - Worst case $O(\log \log n)$ time.
- Cuckoo Hashing
  - Expected $O(1)$ time per insertion.
  - Worst case $O(1)$ time per lookup.

3 Cuckoo Hashing

Cuckoo Hashing is an improvement over two choice Hashing. Let's consider the cells of the Hashing table as vertices in a graph. If an item is hashed to two cells, we draw an edge between those two

[1] Amortized time for insertion

Figure 1: Cuckoo Hashing insertion.

vertices, since that item could be stored in either of those two cells. Also, we only store one item per cell in this setup.In this way, the Hash table can be represented as a multi-graph.

Let n denote the number of vertices and m denote the number of edges (items). Further, we assume $m = n/100$.

In the remaining part of this section, we'll describe the algorithm of Cuckoo Hashing and we'll analyze its running time in the next section.

Lookup To find an item with key w in the Hashing table, we first compute its two hash values $h_1(w), h_2(w)$ and check the corresponding vertices v_1, v_2 in the graph. If one of them contains w, then return that cell.

Insert We first compute the two hash values and check both vertices:

1. If either vertex is empty, place item there.
2. If both full, pick a side with item w' and place w in that cell. Then, insert w' again.
3. If get a cycle, rebuild the whole Hash table.

An example of insertion is shown in Figure 1.

4 Analysis

4.1 Lookup

For each lookup, we only check two vertices. Hence, the worst case time cost is $O(1)$.

4.2 Insertion

There are two parts of time costs for insertion: (1) the time for moving items along the edges, and (2) the time for rebuilding everything if there exists a cycle.

We first analyse the second part. With random Hash functions, the graph becomes a random multi-graph with n vertices and m edges. The probability of existing a cycle in a random graph is

$$\begin{aligned}
\mathbb{P}[\exists \text{ cycle}] &\leq \sum_{k=2}^{m} \mathbb{P}[\exists \text{ length } k \text{ cycle}] \\
&\leq \sum_{k=2}^{m} \underbrace{\frac{\binom{n}{k} \cdot k!}{2k}}_{\#\text{ k cycles}} \cdot \underbrace{\left(\frac{m}{\binom{n}{2}}\right)^k}_{\text{Prob. of one k-cycle}} \\
&\leq \sum_{k=2}^{m} \left(\frac{n^k}{k}\right) \cdot \left(\frac{4m}{n^2}\right)^k \\
&\leq \sum_{k=2}^{m} \frac{1}{k} \left(\frac{4m}{n}\right)^k \\
&\leq \frac{8m^2}{n^2} && \text{(The sum is dominated by the } k = 2 \text{ term.)} \\
&< \frac{1}{100}
\end{aligned}$$

Hence, the probability that there is a cycle in a random graph is bounded by $\frac{8m^2}{n^2}$. In Cuckoo Hashing, if we have $100m$ cells to save m items, the probability of rebuilding the whole graph is small.

Then, we consider the first part, which is the number of times we move items along the edges until we find an empty cell. It's easy to see that the expected insertion time is upper bounded by the expected size of connected component (CC) in the graph. We'll prove that the size of CC is small for random graph.

Claim 1. *In a random graph with n vertices and m edges, for any vertex u,*

$$\mathbb{E}[\textit{size of CC containing } u] = O(1).$$

Before proving Claim 1, let's consider the Erdős-Renyi model $G(n, p)$ first. In this model, each edge is included in the graph with probability p independently. Then, the size of connected component in $G(n, p)$ can be characterized by the Galton-Watson branching process [WG75]. For vertex u, there are $n - 1$ possible neighbors, each of them with probability p. Then, for each v connected with u, there are again $n - 1$ possible neighbors. In this process, we may count the same vertex multiple times but this can only make the size larger. We can view this process as an infinite tree and let C denote the expected size. Then, it's not hard to write the following recursive formula:

$$C = 1 + (n - 1)p \cdot C,$$

which gives $C = \frac{1}{1-(n-1)p}$. For $p = \frac{m}{\binom{n}{2}}$, we have $C \sim \frac{1}{1-\frac{2m}{n}} = O(1)$.

Now, let's get back to the Claim.

Proof of Claim 1. We can upper bound the size of CC as follows:

$$\begin{aligned}
\mathbb{E}[\text{size of CC containing } u] &= 1 + \sum_{v \in V} \mathbb{P}[u \sim v] \\
&= 1 + (n-1)\,\mathbb{P}[u \sim v] \qquad (u \sim v \text{ denotes } u \text{ is connected to } v) \\
&\le 1 + (n-1) \sum_{k=1}^{n-1} \mathbb{P}[d(u,v) = k] \qquad (d \text{ denotes the length}) \\
&\le 1 + (n-1) \sum_{k=1}^{n-1} \binom{n-2}{k-1} (k-1)! \left(\frac{m}{\binom{n}{2}} \right)^k \\
&\le 1 + (n-1) \sum_{k=1}^{n-1} (n-2)^{k-1} \left(\frac{2m}{n^2} \right)^k \cdot e \\
&\le 1 + (n-1) \sum_{k=1}^{n-1} \left(\frac{2m}{n} \right)^k \cdot \frac{e}{n} \\
&\le 1 + e \sum_{k=1}^{n-1} \left(\frac{2m}{n} \right)^k \\
&= O(1)
\end{aligned}$$

□

Then, we can combine these two parts together. Let T be the time for an insertion. Then, we have

$$\mathbb{E}[T] \le O(1) + \frac{1}{100}\,\mathbb{E}[T],$$

which means $\mathbb{E}[T] = O(1)$. Therefore, expected time of Cuckoo Hashing's insertion is $O(1)$.

5 Random Hash Functions

To achieve the random graph, we need fully random Hash functions, which require too much randomness. For alternative, we can use the "universal" Hash functions or k-wise independent Hash functions, which can save randomness while having the same running time for Hashing algorithms. We define them as follows:

Definition 2 (Universality)**.** *Let $\mathcal{H}$ be a Hash family of Hash functions $h : U \to M$. $\mathcal{H}$ is universal if for all $x, y \in U$,*

$$\mathbb{P}_{h \sim \mathcal{H}}[h(x) = h(y)] = \frac{1}{M}.$$

Definition 3 (k-wise independence)**.** *For the Hash family $\mathcal{H}$ defined above, it is k-wise independent if for all $x_1, \ldots, x_k \in U$, $v_1, \ldots, v_k \in M$,*

$$\mathbb{P}_{h \sim \mathcal{H}}[h(x_i) = v_i \ \forall i \in [k]] = \frac{1}{M^k}.$$

It's easy to see that 2-wise independence implies universality.

Consider the Hashing algorithms. For the standard one choice Hashing, we can use universal Hash functions to get $O(1)$ expected time. But in order to have worst case $O(\frac{\log n}{\log \log n})$ time, we need to use $O(\frac{\log n}{\log \log n})$-wise independent Hash functions. For Cuckoo Hashing, we can use $O(\log n)$-wise independent Hash functions to achieve the same running time as fully random hash functions. It can also be shown that 5-wise independence is not enough.

References

[WG75] Henry William Watson and Francis Galton. On the probability of the extinction of families. *The Journal of the Anthropological Institute of Great Britain and Ireland*, 4:138–144, 1875.