Design and Analysis of Algorithms
Massachusetts Institute of Technology 6.046J/18.410J
Debayan Gupta, Aleksander Madry, and Bruce Tidor Handout 8

Recitation 8: Hashing and Streaming Algorithms

1 Tail Inequalities

As we saw in class, when items are inserted into a hash table, they are placed into buckets according to the value of a hash function. Under nearly all hashing schemes, the efficiency of hashing depends on minimizing the number of items which hash into the same bucket, or hash collisions.

Let us consider a simplified model of a hash table with n items and n bins, where the items hash randomly and mutually independently to buckets (For now, we won't worry about the hash function that we use to accomplish this). We can represent this situation as a 'balls-in-bins' model, where each of n balls are independently and equally likely to land in each of n bins. We would like to analyze the number of balls which land in each bin (i.e. the number of collisions).

First, we can consider the average number of balls in each bin.

Observation Let X_b the random variable that represents the number of balls in an arbitrary bin $b \in B$. Then

$$\mathbb{E}[X_b] = \sum_{a \in A} \Pr[a \text{ will end up in } b] = \sum_{a \in A} \frac{1}{n} = 1$$

This makes intuitive sense! On average, we expect each bin to have one ball since the balls are evenly distributed. But what if we are interested the maximum number of balls that we expect to see in a bin? This is relevant if we want to place probabilistic bounds on the *worst-case* cost of operations in our hash table.

To analyze this question, we return to the concept of tail bounds, which we briefly discussed last week. Recall that a tail bound is an upper bound on the probability that a random variable is 'far' from its expected value. In this case, we will attempt to upper bound the probability that there are many balls in one bin. We first try to use the Markov Inequality which was introduced last week.

Markov Inequality

Let Y be an arbitrary discrete random variable that takes non-negative values in the subset Ω of $\mathbb{N}$. Let also $a \geq 0$ then

$$\Pr[Y \geq a] \leq \frac{\mathbb{E}[Y]}{a}$$

The expected number of balls in an arbitrary bin b is $E[X_b] = 1$ so $\Pr[X_b > a] \leq 1/a$. Therefore if you look at every bin separately then the probability that it will have many balls is very small.

But if we want to bound the maximum then we can only use the union bound

$$\Pr\left[\max_{b\in B} X_b \geq a\right] \leq \Pr\left[\bigcup_{b\in B} X_b \geq a\right] \leq \sum_{b\in B} \Pr\left[X_b \geq a\right] \leq \sum_{b\in B} \frac{1}{a} = \frac{n}{a}$$

So the only bound that we can get is that the maximum number of balls in bin is bounded by n, but this is obvious. So our analysis is not tight. But can we do better using Markov's inequality?

Consider the distribution that puts all the balls in one random bin. For this distribution our computation of the expectation still holds! So above analysis that is based only in the expectation of the distribution is actually tight. But obviously the distribution that chooses one bin at random for every ball separately achieves better balance than the distribution that chooses the same random bin of every ball. Therefore we conclude that we need to look at more properties of the distribution than just the expectation to give better values.

Chebyshev's Inequality

Theorem 1 *Let again Y be an arbitrary discrete random variable with expected value $\mathbb{E}[Y]$ and variance $\mathrm{Var}[Y]$. Let also $a \geq 0$ then*

$$\Pr[|Y - \mathbb{E}[Y]| \geq a] \leq \frac{\mathrm{Var}[Y]}{a^2}$$

So in this property we use both the expectation and the variance of the distribution.

The variance of a random variable can be defined in two ways,

$$\mathrm{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] \tag{1}$$
$$\mathrm{Var}[X] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \tag{2}$$

Equation 1 describes the variance as the average squared deviation from the mean of the random variable. Equation 2 describes the variance as the difference between the expected value of X^2 and the square of the expected value of X (be sure you understand why these are different!). These definitions are equivalent, but one may be more convenient to use than the other in any specific case.

Proof. Using Markov's inequality we have

$$\Pr((Y - \mathbb{E}[Y])^2 \geq a^2) \leq \frac{\mathbb{E}[(Y - \mathbb{E}[Y])^2]}{a^2}$$

And by the definition of the variance we have

$$\Pr[|Y - \mathbb{E}[Y]| \geq a] \leq \frac{\mathrm{Var}[Y]}{a^2}$$

□

In order to use this inequality to the balls in bins model we have to compute $\mathrm{Var}[X_b]$. To do so we observe that $X_b = \sum_i X_{b,i}$ where $X_{b,i}$ is the random variable that indicates if the ball i is in the bin b, i.e. $X_{b,i} = 1$ if the ith ball ends to the bth bin and $X_{b,i} = 0$ otherwise. Rewriting X_b in terms of these $X_{b,i}$ is useful because the variance of the sum of uncorrelated random variables is equal to the sum of the variances. This only holds if all variables are pairwise independent.

Each $X_{b,i}$ follows a Bernoulli distribution with $p = 1/n$: thus, $\mathrm{Var}[X_{b,i}] = \frac{1}{n}(1 - \frac{1}{n})$. Putting this all together, we find that the variance of X_b is

$$\mathrm{Var}[X_b] = \mathrm{Var}\left[\sum_i X_{b,i}\right] = \sum_i \mathrm{Var}[X_{b,i}] = \sum_i \frac{1}{n}\left(1 - \frac{1}{n}\right) \leq 1 - \frac{1}{n} \leq 1$$

So now we can use Chebyshev's inequality and we get

$$\Pr[|X_b - 1| \geq a] \leq \frac{1}{a^2}$$

So for $a = c\sqrt{n}$ we get that the $\Pr[|X_b - 1| \geq c\sqrt{n}] \leq \frac{1}{c^2 n}$ and therefore

$$\Pr\left[\left|\max_{b \in B} X_b - 1\right| \geq a\right] = \Pr\left[\max_{b \in B} |X_b - 1| \geq a\right] \leq \Pr\left[\bigcup_{b \in B} |X_b - 1| \geq a\right] \leq$$

$$\leq \sum_{b \in B} \Pr\left[|X_b - 1| \geq a\right] \leq \frac{n}{a^2}$$

Therefore for $a = c\sqrt{n}$ we have

$$\Pr\left[\left|\max_{b \in B} X_b - 1\right| \geq a\right] \leq \frac{1}{c^2}$$

So we can choose $c = 2$ and we will have that with probability at least $3/4$ the maximum load of a bin it will be at most $2\sqrt{n}$. More generally we know that the maximum balls in any bin will be $O(\sqrt{n})$ with constant probability and we can make this constant arbitrarily large.

The reason that Chebyshev's inequality works is that it considers both the expected value and the variance of the distribution and therefore it gets a more tight bound, especially since in this case we are able to use the fact that the indicator variables are pairwise independent. There is again a counter example to show that Chebyshev's inequality cannot give me any better bound.

Chernoff Bound

To get an even tighter bound, we can use the fact that all the random variables exhibit the property of full mutual independence. This is a stronger assumption that pairwise independence! There are groups of random variables which are all pairwise independent but which are *not* mutually independent. Wikipedia has a good example demonstrating this behavior.

Theorem 2 *Multiplicative Chernoff Bound*
Suppose $X_1, \ldots, X_n$ are independent random variables taking values in $\{0, 1\}$. Let X denote their sum and let $\mu = \mathbb{E}[X]$ denote the sum's expected value. Then for any $\beta > 0$,

- $\Pr[X > (1+\beta)\mu] < e^{-\beta^2\mu/3}$, *for* $0 < \beta < 1$
- $\Pr[X > (1+\beta)\mu] < e^{-\beta\mu/3}$, *for* $\beta > 1$
- $\Pr[X < (1-\beta)\mu] < e^{-\beta^2\mu/2}$, *for* $0 < \beta < 1$

Indeed, the requirements for using the Chernoff bound are met. So, we get:

$$\Pr\left[X_b \geq (1+\beta)\mathbb{E}[X_b]\right] \leq e^{-\beta \cdot \mathbb{E}[X_b]/3} \Leftrightarrow \Pr\left[X_b \geq (1+\beta)\right] \leq e^{-\beta/3}$$

For $\beta = 3 \cdot c \cdot \ln n$, we get that:

$$\Pr\left[X_b \geq (1+\beta)\mathbb{E}[X_b]\right] \leq e^{-\beta \cdot \mathbb{E}[X_b]/3} \Leftrightarrow \Pr\left[X_b \geq (1+\beta)\right] \leq \frac{1}{n^c}$$

So, for $c > 2$ after doing a union bound over all the n events X_b for the corresponding bins, we get that the probability that at least one bin has more than $1 + 6\ln n$ balls is at most $1/n$. Thus, the maximum number of balls in any bin is $O(\log n)$ with high probability.

2 Streaming Algorithms

In this recitation, we will consider streaming algorithms which operate on a graph $G = (V, E)$. In general, we assume that we know that the set of nodes V ahead of time and that we have $\Omega(|V|)$ and $o(|V|^2)$ memory: thus, we can store the set of nodes, but we cannot store all the edges of a dense graph. Instead, we process the list of edges as a stream.

When working with large and dense graphs, it would be very convenient if we could somehow reduce the size of the set of edges while preserving certain properties of the graph. We will now examine two streaming algorithms that can find such smaller graphs for two different properties.

Graph Spanners

Our first goal will be to reduce the size of a graph while approximately preserving the distances between every pair of nodes. This produces a *spanner* of the graph.

Definition 1 *Given $G = (V, E)$, a subgraph $H = (V, E_H)$ is a α-spanner of G if $E_H \subseteq E$, and*

$$d_G(u, v) \leq d_H(u, v) \leq \alpha \cdot d_G(u, v) \qquad \forall (u, v)$$

where $d_G(u, v)$ is the length of the shortest path between u and v in G, and $d_H(u, v)$ is the same measure in H.

SPANNER

```
1: E_H = ∅
2: for (u, v) ∈ E do
3:     if d_H(u, v) ≥ α then
4:         E_H = E_H ∪ {(u, v)}
5:     end if
6: end for
7: return E_H
```

A very simple streaming algorithm for constructing an α-spanner in an unweighted graph is given in SPANNER

We can show that this procedure yields an α-spanner of G. Consider any pair of vertices u, v. Clearly $d_G(u, v) \leq d_H(u, v)$ since $E_H \subseteq E$. Now, assume the shortest path $P_{u,v}$ from u to v in G is of length k, which we decompose as $P_{u,v} = \{(u, p_1), (p_1, p_2), \ldots, (p_k, v)\}$, where the p_i are intermediate vertices on the path. We can bound $d_H(u, v) \leq \sum_{(l,h)\in P_{u,v}} d_H(l, h)$.

Now, for each edge $(l, h) \in P$, either this edge is in E_H, in which case $d_H(l, h) = 1$, or $d_H(l, h) < \alpha$ because otherwise we would have kept the edge (l, h) in E_H. Therefore, $d_H(u, v) \leq \sum_{(l,h)\in Pu,v} \alpha = k\alpha \leq \alpha d_G(u, v)$, and H is an α-spanner of G.

To bound the number of edges in H, we observe that H has no cycles of length less than $\alpha + 2$. It turns out by a pretty complex argument that such a graph has at most $n^{1+\frac{2}{\alpha+1}}$ edges. So, if our original graph was dense and we set $\alpha > 1$, we obtain an asymptotically smaller spanner.

SPANNER makes one pass over the stream of input edges. The total amount of memory used is equal to the memory for the nodes plus the memory for the subset of edges in E_H, which is $O(n^{1+\frac{2}{\alpha+1}})$ total. To process each edge, we have to recalculate the distance between the endpoints of that edge in H, which takes $O(|V| + |E_H|) = O(n^{1+\frac{2}{\alpha+1}})$ time.

Graph Sparsification

Note: This section is a little complicated. Don't get hung up on the definitions or lemmas related to sparsification. The main takeaway from this algorithm is the approach used to divide the input stream into sections and obtain an answer by combining the sections without concurrently keeping most of the sections in memory. If you understand what is happening in Figure 1, you've already grokked the important points.

Now we will see how to decrease the size of our graph when we want to preserve capacity of every cut of a graph. For an undirected weighted graph, the capacity is simply the sum of the weights of all the edges that cross the cut. A valid sparsifer of a graph contains a (possibly reweighted) subset of the edges of the original graph which approximately preserves the capacity of all cuts.

Definition 2 *For a cut* $(A, V \setminus A)$*, the capacity* $\lambda_A(G) = \sum_{\{(u,v)\in E|u\in A, v\in V\setminus A\}} w_{(u,v)}$

Definition 3 *A weighted subgraph graph H is a $(1+\epsilon)$-sparsifier (for some $\epsilon < 1$) of G if*

$$(1-\epsilon)\lambda_A(G) \leq \lambda_A(H) \leq (1+\epsilon)\lambda_A(G), \qquad \forall A \subset V$$

There are nonstreaming algorithms for constructing $(1+\epsilon)$-sparsifiers with at most $\epsilon^{-2}n$ edges that take $O(|E|)$ memory. Unfortunately, if G is large enough that we want to find a sparsifier, $O(|E|)$ memory may be enormous! Instead, we shall use this nonstreaming algorithm as a subroutine to construct a streaming algorithm which never has to consider the entire edge set at once. Let A be any such nonstreaming algorithm which constructs a $(1+\gamma)$-sparsifier with at $O(\gamma^{-2}n)$ edges.

To construct our algorithm, we need two key properties of graph sparsifiers.

Lemma 3 *If G_1 and G_2 are two edge-disjoint graphs over a set of vertices V, and H_1 and H_2 are $(1+\epsilon)$-sparsifiers of G_1 and G_2 respectively, then $H_1 \cup H_2$ is a $(1+\epsilon)$-sparsifier of $G_1 \cup G_2$.*

Intuitively, if we independently find sparsifiers for two parts of a graph, we can put them together to obtain a sparsifier for the entire graph.

Lemma 4 *If G_1 is an α-sparsifier of G and G_2 is an β-sparsifier of G_1, then G_2 is an $\alpha\beta$ sparsifier of G.*

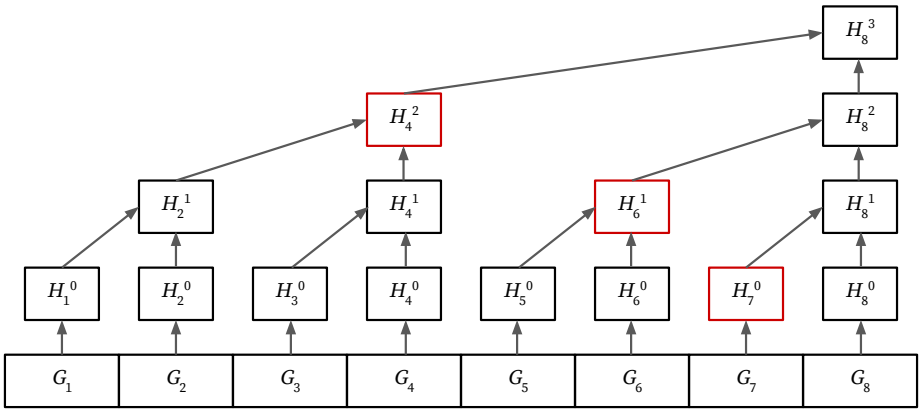
Intuitively, a sparsifier of a sparsifier of a graph is a (slightly less accurate) sparsifier for the original graph.

This algorithm is conceptually very similar to a divide and conquer approach. We will read an input stream of the edges in G in chunks of size $\Theta(n)$. Because $|E| = O(n^2)$, there are at most $O(n)$ such chunks. Let the subgraph defined by the ith chunk of edges be designated G_i. Finding $A(G_i)$ with some fixed γ parameter on each subgraph yields H_i^0, a γ-sparsifier for the given subgraph.

What if we just took the union of all these sparsifiers $\bigcup_i H_i^0$ as our final H? By lemma 3, we know that H is a γ-sparsifier of G, but the number of edges in H might be $\gamma^{-2}n \cdot n$, which is not any better than the original graph! Can we reduce the number of edges in H? By lemma 4, we could calculate $A(H)$ to find a smaller and slightly less accurate sparsifier, but we can't even construct H to pass into A because it would take $O(n^2)$ memory.

Instead, we're going to build up H iteratively while keeping its size manageable. For convenience, we'll assume that the number of chunks t of the edge list is a power of 2. We will have $\log t$ layers of sparsifiers H_i^j, which we will recursively build by sparsifying the union of two H_i^{j-1} from the layer underneath. The computation graph is visualized for $t = 8$ in Figure 1.

Figure 1: Computation graph for $t = 8$



Formally, we define

$$H_i^0 = A(G_i) \qquad i \in \{1, \ldots, t\}$$
$$H_i^j = A(H_{i-2^{j-1}}^{j-1} \cup H_i^{j-1}) \qquad i \in \{1 \cdot 2^j, 2 \cdot 2^j, 3 \cdot 2^j, \ldots, t\}, j \in \{1, \ldots, \log t\}$$

By lemmas 3 and 4, H_i^j is a γ^j-sparsifier of the subgraph defined by

$$\bigcup_{k=(i-1)2^j+1}^{i2^j} G_i$$

Thus, $H_1^{\log t}$ is a $\gamma^{\log t}$-sparsifier of $\bigcup_{k=1}^{t} G_k = G$. If we set $\gamma = \epsilon/(2 \log t)$ we obtain a $(1+\epsilon)$ sparsifier of G with $O(\epsilon^{-2} n \log^2 n)$ edges.

Does this algorithm really take $o(n^2)$ space? It certainly looks like we have to store a lot of intermediate H_i^j. But because of the order in which we merge the sparsifiers, we can actually discard $H_{i/2}^{j-1}$ and H_i^{j-1} after calculating H_i^j. For instance, after we process G_1 we calculate and store H_1^1. After we process G_2 we calculate H_2^1, and then calculate and store H_1^2 and discard H_1^1 and H_2^1. Thus, the maximum number of H_i^j we ever store is $\log t$ after step $t-1$ (the nodes highlighted in red in figure 1, and the total memory usage is $O(\gamma^{-2} n \log t) = O(\epsilon^{-2} n \log^3 n)$.

3 Appendix: Open Addressing

Another alternative to dealing with collisions in a hash table where the set of keys can be dynamic is open addressing. We only allow one element in each bucket: if a collision occurs, we move one element out of its 'proper' bucket and place it in a different bucket. This implies that there can only be as many elements in the table as there are buckets, and thus that $\alpha \leq 1$. The details of how the new bucket is selected differentiate open addressing algorithms.

While open addressed hash tables generally have asymptotic performance similar to or even worse than chained hash tables, in practice they can be both more memory efficient (because no pointers or data external to the table itself must be stored) and faster (because following pointers in a linked list is cache unfriendly). We will briefly describe two common variants of open addressing: linear probing, and double hashing.

Both work in the same basic way. Because we may need to probe several buckets in order to find our key, the hash function h is actually both a function of k and the number of probes we have made i.

To search for a key, we perform the following procedure

FIND-KEY(k)

```
1: for i ∈ {0 ... m − 1} do
2:     b_i = h(k, i)
3:     if T[b_i] == k then
4:         return TRUE
5:     else if T[b_i] == null then
6:         return FALSE
7:     end if
8: end for
9: return FALSE
```

To make this concrete, let's start by examining **linear probing**. Given a standard hash function $h' : U \to \{1, \ldots, m\}$ and a table of size m, we construct

$$h(k, i) = (h'(k) + i) \mod m$$

Now we can examine what FIND-KEY does with linear probing: it calculates $h'(k)$, then probes for k by starting from the expected bucket and walking one bucket at a time forward: at each point, if we ever find k or an empty bucket, we are done. If we traverse the entire table without finding k, we return FALSE.

This simple approach is very cache-friendly (since we examine adjacent array elements) but can lead to problems with key stacking. In particular, we can end up with long runs of continuously

occupied buckets. This occurs because given a series of occupied buckets of length k, the probability that the next bucket after the series is occupied on the next insertion is k/m, compared to a probability of $1/m$ for an empty bucket not directly after any occupied buckets. The longest sequences of occupied buckets grow quickly, leading to poor probing behavior.

One way to address this problem is by using **double hashing**. Instead of probing adjacent buckets, we use a second hash function to determine an offset for each key which we use to probe.

$$h(k, i) = (h'(k) + ih''(k)) \mod m$$

Note that in order to ensure every bucket is eventually probed, we require that $\gcd(h''(k), m) = 1 \forall k$. Double hashing avoids key stacking, but it requires more calculation since we must compute a second hash function at every iteration and is less cache-friendly.

An interesting variant of linear probing is **Robin Hood hashing**. This is a clever way of improving linear probing to mitigate key stacking while preserving cache friendliness. Each time we probe to insert a key k but find its slot already occupied by another key l, we determine whether k is l is further from its 'preferred' probe location where $i = 0$. If l is further than k, we continue probing. If k is further away than l, we replace l with k, then seek to reinsert l into the hash table. This reduces the variance of the length of any probe sequence by seeking to equalize the distance of every element from its ideal position. Thus the name Robin Hood hashing: 'rich' elements which are close to their ideal hash location give up their slots for 'poor' elements which are farther away.