CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 14 – March 2nd, 2023

Prof. Jelani Nelson *Scribe: Nikki Suzani, Eshaan Bhansali*

1 Power of Two Choices

The power of two choices [1] [2] takes two random hash functions h, g that map $[U] \to n$, and finds that the expected max load on each slot goes from $\frac{\ln(n)}{\ln\ln(n)}$ to $\frac{\ln\ln(n)}{\ln(2)} + O(1)$. Note that with $k \geq 2$ hash functions, there is only a constant factor improvement, with the $\ln(2)$ in the denominator changing to $\ln(k)$.

To prove this, we'll do analysis based on height. Given $x \in DB$, we define the height of some x to be its height in the stack at the time it was inserted; for example, if x was the hth item put into its bucket, then its height is h. Our goal is to show that when we get to some height of $\frac{\ln\ln(n)}{\ln(2)}$ the number of elements with that height is less than 1. Since this number must be an integers, this means that we expect no elements with that height, and the expected max load is bounded by $\frac{\ln\ln(n)}{\ln(2)} + O(1)$.

Define B_i to be the number of slots with at least i items. Let's look at the ratio $\frac{B_i}{n}$. If we can show that this fraction is less than $\frac{1}{n}$ we have proven the statement, since it implies there are less than 1 buckets with the given load.

Note that clearly $\frac{B_1}{n} \leq 1$, as we have n slots. Further, if a bucket has height $i+1$, then there is one unique person in the bucket with height $i+1$.

$$B_{i+1} \leq \sum_x \mathbf{1}\{\text{height}(x) \geq i+1\}$$

We can now take the expectation.

$$\begin{aligned}
\mathbb{E}[B_{i+1}] &\leq \sum_x \mathbb{E}\{\text{height}(x) \geq i+1\} && (1)\\
\mathbb{E}[B_{i+1}] &\leq \sum_x \mathbb{P}\{\text{height}(x) \geq i+1\} && (2)\\
\mathbb{E}[B_{i+1}] &\leq n * \mathbb{P}(\text{height}(x) \text{ for some } x \geq i+1) && (3)\\
&= n * \mathbb{P}(\text{height(x)}) \geq i+1 && (4)\\
&= n\left(\frac{B_i}{n}\right)^2 && (5)\\
&= \frac{B_i^2}{n} && (6)
\end{aligned}$$

Here, (5) comes from the fact that all these B_i are random variables related to each other, based on the same hash functions. Pretend the previous B_j for $j \in [1, i]$ are fixed. For the height to be at least $i + 1$, it means that for each of the places x hashed, each of those places had to have a load of at least i. We know that the probability a hash function takes us to a place with load at least i is $\frac{B_i}{n}$, so we can substitute that in here.

Dividing both sides by n, we see that

$$\mathbb{E}\left[\frac{B_{i+1}}{n}\right] \leq \left(\frac{B_i}{n}\right)^2$$

.

This analysis is a little hand-wavy, since we already considered the randomness. However, the key is that if things are going according to their expectation, then the fraction of buckets that have load i is decreasing in this way.

We know that $\frac{B_2}{n} \leq \frac{1}{2}$ with probability 1, by counting the places where elements can hash. If things go according to their expectation, then generally

$$\frac{B_{2+j}}{n} \leq \frac{1}{2^{2^j}}$$

To solve, we want $\frac{B_i}{n} < \frac{1}{n}$.

$$\frac{1}{2^{2^j}} < \frac{1}{n}$$

$$j \geq \log\log(n)$$

Another paper from 2003 [3], takes d hash functions and splits up a hash table of n slots into d buckets with $\frac{n}{d}$ slots. Here, you have a hash function for each bucket, such that when you see an item you hash it to its slot in each bucket and then put it into the least loaded bucket. (Ties are broken by putting it into the leftmost least-loaded bucket.) Through this approach, with high probability the max load is $\frac{\ln\ln(n)}{d \cdot \phi_d} + O(1)$ where ϕ_d is a sequence of numbers that are in the range of $[1.61, 2]$. This does better than the power of two choices, since it divides by d instead of $\ln(d)$.

2 An Aside on Upcoming Lectures

Next lecture we'll talk about spectral graph theory. This idea comes from adjacency theory, where you are able to study properties of the graph from spectral properties of the matrix. For example, given the eigenvalues and eigenvectors of the adjacency matrix of the graph, you can know how many connected components are in the graph.

We'll later talk about linear programming and how to solve linear programs, looking at the details of the Simplex method and Interior-Point polynomial time algorithms.

3 Online Algorithms

Online algorithms are about decision-making in the face of uncertainty about the future. That is, without knowing the future, the goal is to make the best decision (or close to the best decision)"on-the-fly." We then compare our results with an omniscient being who knows the future.

4 Pot of Gold

Let's start with the **Pot of Gold** problem. Imagine a long hallway that contains equally-spaced treasure chests on both sides. The distance between all the treasure chests is one yard. We know that one of the treasure chests has gold inside, and the rest are empty. Each timestep, we walk to a new chest and open it to see if the gold is inside. The goal is to walk as few yards as possible to find the gold.

Let's say the gold is at some position t. We know that OPT will pay t yards, since it goes directly to where the gold is.

To compete with OPT, let's start by trying zig-zagging, going from -1 to 1 then -2 to 2. This leads to a lot of work taken to walk between the sides, so perhaps we should consider spending more time on each side and checking nearby chests.

One method to do this is going through powers of two, from $-1 \to 2 \to -4 \to 8 \to 16 \to 32$. The reason this works is because we need to walk to the origin regardless when crossing sides, so the time to go even further on the other side is amortized a similar amount.

Say $t \in [2^m, 2^{m+1}]$. We pay $2 \cdot (1 + 2 + 4 + ...2^k) + |t|$, considering the time it takes to go back to the origin. In the worst case, $k = m + 1$, since we got almost up to t on one side, but went across and took about 2t additional time to get to t, so we have

$2 \cdot \underbrace{(1 + 2 + 4 + ...2^{m+1})}_{4t} + |t| \leq 9t.$

Definition 4.1. For any algorithm A, it is C-competitive if for all inputs σ,

$$\text{cost}(A(\sigma)) \leq C \cdot OPT(\sigma) + O(1)$$

Thus, by Definition 4.1, this algorithm is 9-competitive.

5 Ski Rental Problem

Imagine you and your friends are going to a ski resort, but haven't picked an end date for your vacation. Every day you decide whether to go home or continue skiing. The question is about whether you should rent skis each day, or buy the skis (which has a higher fixed cost, but will be helpful if you're staying for a long time).

In this scenario, renting skis is \$1 per day, and buying skis is a one-time cost of \$$b$ skis.

OPT, knowing that we stay for d days, will buy on day 1 if $d \geq b$ and rent every day otherwise. Thus, $OPT = \min\{d, b\}$.

A good strategy for us would be to rent for the first $b-1$ days, then buy on day b. If $d < b$, we pay the same that OPT does. If $d \geq b$, then we pay $\frac{2b-1}{b}$, so we're competitive with OPT here (we are approximately 2-competitive since $\frac{2b-1}{b} \approx 2$).

6 List Update Problem

Let's explore another online algorithm problem, which is a warm-up for understanding paging and cache update problems. Here, the goal is to maintain a linked list of items with three kinds of operations:

- `access(x)`: Start at the beginning of a linked list, and follow list pointers until you get to x. The cost of this is the position of x.
- `insert(x)`: Append x to the end of the linked list, and pay the length of the list.
- `delete(x)`: Walk to x, then remove it, and pay the cost of the position of x.

Note that at either `insert` or `access`, after performing the operation you can choose to bring x closer to the front by any number of positions for free.

The goal is to reduce the cost by deciding at each `access` or `insert` whether to bring x to the front or not.

There are a few heuristics here that are natural:

- **MF** or **Move-to-Front**. That is, each time you `access` or `insert` x, you move it all the way to the front.
- **Transpose**. Here, each time you `access` or `insert` x, you bring x one closer to the front.
- **FC** or **Frequency Count** which tries to keep item in decreasing access of frequency, based on past `accesses`. An item goes up in frequency when you `access` it, so once accessed you know where to move it to maintain the sorted order.
- **SFC** which stands for **Static Frequency Count**. Here, you look into the future and keep items in decreasing order of the final frequency of their `accesses`. Note that this is the best static ordering. Since we can't look into the future, we can't run this algorithm, but we can use it to compare against.

The first paper [4] to not make any assumptions about the frequencies of possible xs found that if items are initially sorted by time of first access, then for all sequences of operations σ, cost(**MF**(σ)) $\leq 2 \cdot$ cost(**SFC**(σ)). This showed **MF** is statically optimal.

What's more interesting is that **MF** is not just statically optimal, but also dynamically optimal. Let's look at a model where transpositions cost 1 when moving something other than x toward front, and 0 cost when moving x.

Theorem 6.1. *$\forall A, \sigma$ where A is any algorithm making decisions (including OPT), and σ is the sequence of operations,*

$$cost(\boldsymbol{MF}(\sigma)) \leq 2 \cdot cost(A(\sigma)) + \underbrace{P(A(\sigma))}_{\text{paid transpositions cost}} - \underbrace{F(A(\sigma))}_{\text{\# of free moves}} - m. \ [5]$$

*Note that this assumes A and **MF** start with the same ordering (possibly the empty list).*

Proof. Let's use a potential function argument where Φ(State) = the number of inversions in **MF**'s list, according to the ordering in A's list.

As a reminder, the Φ-cost of an operation = Total cost + $\Delta\Phi$, meaning Total Φ-cost = Total cost + Φ(final) - Φ(initial).

Thus, the Total cost = Total Φ-cost + Φ(initial) - Φ(final).

Since we start with the same number of inversions, $\Phi(\text{initial}) = 0$ and $\Phi(\text{final}) \geq 0$ and this then means that Total cost $\leq$ Total Φ-cost.

Now, let's look at the orderings of both **MF** and A. Suppose we are accessing element x, which is in position k in **MF** and in position i in A. There are some t items that are before x in **MF** and after x in A (the red items in Figure 1). All other non-x items in **MF** which are before x, must be also before x in A (the blue items in Figure 1). Note that the number of elements in this set are $k - t - 1$, since there are t elements that are before x in **MF** and A, x is one element, and there are k total elements up to x.

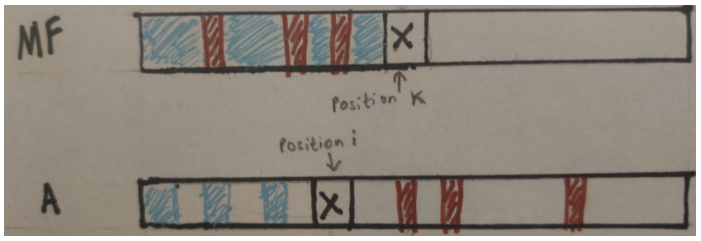

Figure 1: Comparison of lists using algorithm A and **MF**.

Let's solve for the Φ-cost here. We know that the actual cost of an operation = k. The $\Delta\Phi$ here considers the difference between A's movements and **MF** moving x all the way to the front. When **MF** moves x all the way to the front, the t items are now no longer inversions, and the $k - t - 1$ elements before x in **MF** and A now become inversions. Thus, $\Delta\Phi = (k - t - 1) - t = -2t + k - 1$. Thus, the Φ-cost is $k + -2t + k - 1 = 2(k - t) - 1$. We know that $k - t - 1 \leq i - 1$, since there are only $i - 1$ items before x in A's list. Then, the total cost $\leq 2i - 1$. Then, summing this over all elements we get $2 \cdot \text{cost}(A(\sigma)) - m$. And knowing that when A does a free move to the front it only helps Φ by costing 1 to A, we have completed the proof.

□

References

[1] Azar, Broder, Karlin, Upfal. Balanced Allocations (extended abstract) *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[2] Michael Mitzenmacher, Andrea W. Richa, Ramesh Sitaraman. Chapter 9: The Power of Two Random Choices: A Survey Of Techniques And Results. *Handbook of Randomized Computing.* 2001. Kluwer Academic Publishers.

[3] Berthold Vocking. How asymmetry helps load balancing. *J. ACM.*, 50(4):568–589, 2003

[4] Jon L. Bentley, Catherine C. McGeoch. Amortized Analyses of Self-organizing Sequential Search Heuristics. *ACM*, 28(4):404–411, 1985.

[5] Daniel Sleator, Robert Tarjan. Self-Adjusting Binary Search Trees. *J. ACM*, 50(4):568–589, 2003.