Lecture 7: Perfect Hashing

Readings: CLRS section 11.5

Lecture Overview

1. Universal hashing constructions
2. The static dictionary problem
3. Perfect hashing

Universal Families - Constructions

Last lecture, we discussed the fact that the collection of all hash functions $h : \{0, \ldots, u-1\} \mapsto \{0, \ldots, m-1\}$ forms a UHF. We now discuss in detail one construction, much smaller in size. We also present a second construction without proof.

What do we mean by "smaller in size"? As discussed, we need to be able to store, in some format, a specification of the particular function $h \in \mathcal{H}$. Assuming that we can store keys from $\mathcal{U}$ in one word, we would like to be able to specify hash functions with $O(1)$ words as well.

The dot-product family

Assumptions

- m is a prime
- $u = m^r$ where r is an integer

We want m to be prime for reasons we'll see in a bit. What happens if, in our application, m is not a prime? We choose a prime larger than m. We have efficient algorithms which can find a nearby prime. We're not going to cover that. In real cases, we can always round up m and u to satisfy our requirements.

Once we have this, it's going to be pretty natural to view our keys as r tuples with entries in $\{0, \ldots, m-1\}$. So, a key k can be written as $k = \langle k_0, k_1, \ldots, k_{r-1} \rangle$, and $0 \leq k_i < m$.

Our universal family $\mathcal{H}$ will have a function for every r-tuple a. So the size of the family is the same as the size of $\mathcal{U}$. To specify a function $h \in \mathcal{H}$, we just need

to specify its associated a. Given a, its associated hash function will be called h_a. If keys scan be specified in 1 word, then the "names" of the hash functions in $\mathcal{H}$ can also be specified with 1 word!

We need to specify, given a and k, how to compute $h_a(k)$. Similarly to k, we can write a as an r-tuple $a = \langle a_0, a_1, a_2, \ldots, a_{r-1}\rangle$. We then define

$$\begin{aligned} h_a(k) &= a \cdot k \bmod m \text{ (dot product)} \\ &= \sum_{i=0}^{r-1} a_i k_i \bmod m \end{aligned} \tag{1}$$

Then our hash family is $\mathcal{H} = \{h_a | a \in \{0, 1, ..., u-1\}\}$.

It is good to think of r as pretty small/bounded so we can treat it like a constant. Therefore, We're going to assume that you can compute $h_a(k)$ in constant time. As stated above, storing h_a also takes $O(1)$ space.

We call this family the *dot-product hash family*. We now prove that it is universal.

Theorem 1. *The dot-product hash family $\mathcal{H}$ is universal.*

Proof. We want to prove that, if we choose a randomly, the probability of two keys mapping to the same value is at most $1/m$. Fix any 2 distinct keys. Say the keys are $k = \langle k_0, k_1, \ldots, k_{r-1}\rangle$ and $k' = \langle k'_0, k'_1, \ldots, k'_{r-1}\rangle$. If two vectors are different, then they differ in at least one entry. WLOG, assume that $k_0 \neq k'_0$.

What conditions on a do we need for $h_a(k) = h_a(k')$? We must have

$$\begin{aligned} h_a(k) = h_a(k') \text{ iff } & \sum_{i\geq 0} a_i k_i = \sum_{i\geq 0} a_i k'_i \mod m \\ \text{iff } & a_0 k_0 + \sum_{i\geq 1} a_i k_i = a_0 k'_0 + \sum_{i\geq 1} a_i k'_i \mod m \\ \text{iff } & a_0(k_0 - k'_0) = \sum_{i\geq 1} a_i(k'_i - k_i) \mod m \\ & (m \text{ is prime} \Rightarrow \mathbb{Z}_m \text{ has multiplicative inverses}) \\ \text{iff } & a_0 = (k_0 - k'_0)^{-1} \sum_{i\geq 1} a_i(k'_i - k_i) \mod m \\ \text{iff } & a_0 = g(k, k', a_1, \ldots, a_{r-1}) \mod m \end{aligned}$$

Math notes: Multiplicative inverses – for every value x, there exists y such that $xy = 1 \mod m$, given that m is prime.

Using the principle of deferred decisions, we choose the random vector $a = \langle a_0, a_1, \ldots, a_{r-1} \rangle$ by first choosing a random vector $\langle a_1, \ldots, a_{r-1} \rangle$ and then choosing, independently, a random a_0.

By the above derivation, for any k and k', and having picked $\langle a_1, \ldots, a_{r-1} \rangle$, $h_a(k) = h_a(k')$ iff a_0 takes on the unique value $g(k, k', a_1, \ldots, a_{r-1}) \mod m$. This happens with probability $\frac{1}{m}$.

□

Another universal hash family from CLRS: Choose prime $p \geq u$ (once). Define $h_{ab}(k) = [(ak + b) \mod p] \mod m$. We have $\mathcal{H}_{p,m} = \{h_{ab} | a, b \in \{0, 1, ..., p-1\}, a \neq 0\}$. The family depends on the two choices for p and m. Note that, to specify a function from the family, one needs to specify the values of a and b. We assume that this can be done with 2 words. In addition, we may need to also specify m, the size of the hashing table. We will refer to this family as the CLRS UHF.

Applications

The Static Dictionary Problem

As with a Dictionary, a Static Dictionary is an Abstract Data Type (ADT) that maintains a set of items. Each item is associated with a key. This time, however, we assume that the set of keys is a static set so that only the **search** operation is supported.

One can think of many applications. Essentially, any application when the set of keys changes only seldomly.

We would like a solution where searches can be done in $O(1)$ time in the **worst-case**. Further, we would like to use as little space as possible. To store n keys, we would like to use $O(n)$ space in the worst case. Finally, given n arbitrary keys, we would like to have an efficient algorithm that builds our data structure.

Perfect Hashing

In this section we will show how to build, in expected polynomial time (in n), a hashing based solution to the static dictionary problem with $O(1)$ worst-case search time and $O(n)$ worst-case space. That is, we will describe a *Las Vegas* algorithm that builds, given a set of n keys, a self-contained data structure that solves the static dictionary problem using $O(n)$ space which supports $O(1)$-time searches. It is based on efficient UHFs.

In what follows, we will be using UHFs with varying table sizes. For that reason, rather than working with the dot-product family, it is more convenient to work with the CLRS family. We will keep p fixed but vary the size m of the hash table.

An $O(n^2)$ solution

We start by providing a perfect hashing solution to the static dictionary problem that uses $O(n^2)$ space to store any n keys. This solution will provide a key ingredient to our final solution. The idea is to use a UHF with $m = n^2$. Intuitively, the hash table is so sparse that we will expect few collisions. In fact, we claim that, for a random hash from the UHF, the probability that there is a collision is at most one half.

Claim 2. *If we store n keys in a table of size $m = n^2$ using a random hash function from a UHF then, with probability at least $\frac{1}{2}$, there are no collisions.*

Proof. Let the keys be $k_1, \ldots, k_n$. Recall that our probability space is over the set of hash functions from our UHF. Rather than bounding the probability of no collisions, we will show that the probability of at least one collision is at most one half.

For two distinct keys k_i and k_j, let $E_{i,j}$ denote the event that the two keys collide. We are interested in bounding the probability that any two keys collide. In other words, we are interested in bounding the probability of the event $E = \cup_{i \neq j} E_{i,j}$. As we are drawing from a UHF, for any i, j, $\Pr[E_{i,j}] \leq \frac{1}{m} = \frac{1}{n^2}$. Using the Union Bound, we get $\Pr[E] \leq \sum_{i,j} \Pr[E_{i,j}] \leq \binom{n}{2} \frac{1}{n^2} \leq \frac{1}{2}$ □

Exercise: Compare the above proof with the treatment in CLRS of the Birthday Paradox using indicator variables (page 132).

Using claim 2, we can develop a Las Vegas algorithm to create the desired data structure. Keep choosing hash functions from the CLRS UHF until we get no collisions. Each trial is successful with probability at least $\frac{1}{2}$. On expectation, we will be successful after two tries. Also, the probability that we haven't been successful after $\log n$ tries is at most $\frac{1}{n}$.

What is the exact space complexity? We are using exactly $3 + n^2$ space. We use 2 words to store the a and b of our hash function and 1 word to store $m = n^2$. We also use n^2 words for the hash table. Once the keys are stored in the table, any **search** operation requires at most 4 probes to the table: three probes to get the constants a, b and m and the fourth to access the table addressed by $h_{a,b}(k)$. We assume that $h_{a,b}(k)$ can be computed in $O(1)$ time.

The FKS algorithm

We now present an algorithm that reduces the space needed to $O(n)$. The algorithm is due to Fredman, Komlós and Szemerédi (1984). As with consistent hashing, the idea is beautiful in its simplicity. It is based on two ideas. The first is to use a second level of hashing, instead of linked-lists, to deal with collisions. The second relies on the solution with $O(n^2)$ space to choose the appropriate size of the second level of hashes to guarantee no further collisions. A precise analysis ties everything together.

To make things concrete, here is the plan. We will pick a "good" hash h with a table size of $m = n$. For slot j, let n_j be the number of keys that are mapped to j. For slot j, to deal with the n_j collisions, we will use a hash h_j with table size of n_j^2. We will use the following technical claim:

Claim 3. *If we store n keys in a table of size $m = n$ using a random hash function from a UHF, then with probability at least $\frac{1}{2}$, $\sum_j n_j^2 \leq 4n$.*

We will prove the claim later. Before, let's formalize the algorithm for the construction of the static dictionary and analyze its time and space complexity.

FKS Algorithm:
Input: Keys $k_1, \ldots, k_n$
Output: A data structure to store the keys with constant search time.

1. Pick a prime $p > u$ to work with the CLRS UHF.
2. Pick a random hash from the CLRS UHF with $m = n$ until $\sum_j n_j^2 \leq 4n$. Call the function h.
3. For each slot j, pick a random hash from the CLRS UHF with $m = n_j^2$ until there are no collisions. Call the function picked for slot j, h_j.

Let's start by analyzing the amount of used space. The top level hash uses space n. The sum of all the hash tables at the second level is $4n$. In addition, we need three words to store the top level hash h (we need to store a, b and $m = n$). We also need 3 words for each slot, one to store n_j and two to store the parameters a and b of h_j. The total space used is $n + 4n + 3n + 3 = 8n + 3 = O(n)$.

Once the data structure is built, what is the time complexity of a particular search operation? Fix a key k. We need three probes to get a, b and $m = n$. We need $O(1)$ to compute $h_{a,b}(k)$. Say that $h_{a,b}(k) = j$. We need 3 probes to get n_j and the corresponding parameters of h_j. We then need $O(1)$ time to compute $h_j(k)$. We then

make the final probe at location $h_j(k)$ of the j-th table of the second level. All in all, we make 7 probes and use $O(1)$ time.

What is the time complexity of the FKS algorithm? Step #1 can be done in time polynomial in $\log u$. Each trial in step #2 takes linear time and, on expectation, we only need $O(1)$ many trials. Step #3 has to be done for each slot j. Once again, each trial takes time linear in n_j and, in expectation, $O(1)$ many trials suffice. So, in the aggregate, step #3 takes $O(n)$ time. Therefore, aside from the time needed to find p, the algorithm takes expected $O(n)$ time.

We now prove claim 3.

Proof. Recall that n_j denotes the number of keys mapped to slot j. Note that n_j is a random variable. Given fixed keys $k_1, \ldots, k_n$, a random hash function from our UHF defines the variables n_j for $j = 1, \ldots, m$. Recall that we are dealing with the case where $m = n$.

The plan for the proof is to show that $E[\sum_j n_j^2] \leq 2n$ and then use Markov's inequality to conclude that $\Pr[\sum_j n_j^2 \geq 4n] \leq E[\sum_j n_j^2]/4n \leq \frac{1}{2}$. Note that Markov's inequality applies as $\sum_j n_j^2 \geq 0$.

Rather than working directly with $E[\sum_j n_j^2]$, we start with the similar expectation $E[\sum_j \binom{n_j}{2}]$. We then use simple algebra to bound the desired expectation.

What is $E[\sum_j \binom{n_j}{2}]$? Note that each term $\binom{n_j}{2}$ denotes the number of pairs that collide because the keys are mapped to slot j. As we count over all slots j, the summation inside the expectation denotes the total number of pairs that collide! To compute the expected number of collisions, we turn to indicator variables. For a pair of keys k_i and k_j, let $I_{i,j}$ denote the indicator variable for the event that the two keys collide. Then

$$\begin{aligned} E\left[\sum_j \binom{n_j}{2}\right] &= E[\#\text{ collisions}] \\ &= E\left[\sum_{i \neq j} I_{i,j}\right] \\ &= \sum_{i \neq j} E[I_{i,j}] \\ &\leq \binom{n}{2}\frac{1}{n} \end{aligned}$$

To go from $E[\sum_j \binom{n_j}{2}]$ to $E[\sum_j n_j^2]$ we use the identity $n_j^2 = n_j + 2\binom{n_j}{2}$ which can be easily verified. We conclude:

$$
\begin{aligned}
E\left[\sum_j n_j^2\right] &= E\left[\sum_j \left(n_j + 2\binom{n_j}{2}\right)\right] \\
&= E\left[\sum_j n_j\right] + 2E\left[\sum_j \binom{n_j}{2}\right] \\
&\leq n + 2\binom{n}{2}\frac{1}{n} \\
&\leq 2n
\end{aligned}
$$

□