CS648 : Randomized Algorithms
Semester I, 2011-12, CSE, IIT Kanpur

Hashing with linear space and worst case $O(1)$ search time

Before we embark on the journey of a very beautiful result of theoretical computer science, we would like to remind the reader about the RAM (random access memory) model of computation.

Consider any common computer these days. It consists of a processor and memory. The processor executes arithmetic or logical instructions specified in a program. The memory, which we call RAM, stores the data as well as the programs. There is a basic unit of memory, called *word*. Two key features of the RAM and processor are the following :

1. It takes $O(1)$ time to access any instruction or data unit located in the RAM. (this is the reason it is called *random access memory*).

2. Any arithmetic operation whose input as well as output consists of a constant number of words can be computed in $O(1)$ time. This is also conveyed implicitly by the widely used term "64-bit architecture".

The RAM model of computation captures the two fundamental features of ordinary computers. Furthermore, it is assumed that the entire input of the problem is stored in the RAM. We analyse time complexity of an algorithm almost always in this RAM model of computation. However, there are other models of computations as well, namely, Turing machine, comparison tree, decision tree, cell probe model.

Hashing is a very powerful data structure technique which exploits the features of RAM model of computation mentioned above in a very powerful way.

1 What is hashing ?

Hashing offers a very simple and efficient solution to the following problem. There is a set $U = \{0, \cdots, m-1\}$ of integers, called universe, and we are given a set $S \subset U$. The number of elements $s = |S|$ is very very small compared to $m = |U|$. We have to preprocess the set S to build a data structure so that given any element $x \in U$, it can be determined efficiently, preferably in $O(1)$ time, whether $x \in S$. For example, U could be the set of `long` integers in these days 64-bit architecture (thus $m \approx 10^{17}$), but S could consist of a few thousands `long` integers only.

There is a trivial $O(m)$ space solution for the above problem where we keep a Boolean array A such that $A[i] =$ `true` if and only if $i \in S$. In the underlying RAM model, it takes $O(1)$ time to access any location of A, so the search time is $O(1)$. However, this solution is not acceptable due to the unreasonable space requirement. There also exist space efficient solution (binary search tree) which guarantee $O(\log n)$ search time. The aim is to achieve the best of these two obvious solutions. With this objective hashing was invented in 1950's as a heuristic which works very well in practice.

Hashing consists of a hash table which is an array T of size $n = O(s)$ and a hash function $h : U \to \{0, .., n-1\}$ such that it takes $O(1)$ time to compute $h(x)$ for any $x \in U$ in the underlying RAM model of computation. For any $i < n$, $T(i)$ would store a linked list of all those elements $a \in S$ with $h(a) = i$.

In order to search for any element $x \in U$, we compute $j = h(x)$ and scan the list $T[j]$ to determine if x is present there. So the search time will be of the order of the number of elements stored in the list $T[j]$. A hash function h is good for S if it distributes the elements of S *nearly* uniformly in the entire table.

For three decades after its invention in 1950's, hashing remained a very widely used heuristic for search data structure. However, nothing much could be achieved on the theoretical foundation of hashing.

In 1980's, a significant breakthrough was made in the area of hashing and the following result was derived. It is possible to preprocess a given set S in expected $O(s)$ time to build a data structure based on hashing which guarantees worst case $O(1)$ search time and still occupying $O(s)$ space. Interesting, the ingredients of this powerful result were elementary knowledge of primes, mod, and of course elementary probability theory. In this lecture, an attempt has been made to provide a detailed sketch of this result.

2 Preliminaries

We start with the following, somewhat discouraging, observation which essentially says that no hash function h can be good for all subsets $S \subset U$.

Observation 2.1 *For every hash function h, there exists a subset S such that $\min(m/n, s)$ elements from it are hashed to the same index in the hash table.*

The above observation follows from simple pigeon hole principle. If $h(a) = h(b)$ for any given two elements $a, b \in U$, that is, if they are hashed to the same index in the hash table, then we say that a and b collide under the hash function h. The above observation implies that for every hash function there exists at least m/n elements from the universe which collide under h. A hash function is said to be perfect for a set S if no two elements from S collide under h.

Let us consider a very simple hash function h defined by $h(x) = x \bmod n, \forall x \in U$. The following simple observations about mod function provides the conditions when any two elements may collide under this function.

Observation 2.2 *$a \mod \ell = b \mod \ell$ if and only if $|a - b|$ is a multiple of ℓ.*

Our first step is to find out the reasons as to why the above simple hash function works so well in practice.

2.1 Why hashing as a heuristic itself works so well in practice

The answer to this question lies in the uniform probability distribution underlying the set S for most of the applications. But it is worth exploring as to why this uniform probability distribution proves to be so useful for the above hash function. Let $\{x_1, ..., x_s\}$ be the set S formed by selecting a uniformly random subset of s elements from U. Let Y_i be the random variable for the number of elements from $S \backslash \{x_i\}$ which are hashed to the same location as x_i. Our objective is to calculate $\mathbf{E}[Y_i]$. We can express Y_i as sum of $s - 1$ random variables Y_{ij}'s $i \neq j$ such that $Y_{ij} = 1$ if $h(x_i) = h(x_j)$. Conditioned on x_i taking a value a, it follows from Observation 2.2 that $h(x_i) = h(x_j)$ if and only if x_j takes values which is away from a by nonzero multiples of n. The set of such possible values are $\lceil m/n \rceil - 1 \leq \frac{m-1}{n}$, whereas the total possible choices for x_j is $m - 1$. Hence

$$\Pr_{S \subset_r U}[h(x_i) = h(x_j] \leq \frac{1}{n} \tag{1}$$

As a result, the expected number of elements from S colliding x_i is $\mathbf{E}[Y_i] = \sum_{i \neq j} \mathbf{E}[Y_{ij}] = \frac{s-1}{n}$.

So, the expected number of elements colliding a given key $x_i \in S$ is less than 1 when $n \geq s$ and the set S is formed randomly uniformly. Hence the expected search time for any key will be constant. This explains why the simple hash function works so well due to the uniform random distribution of the set S.

How can the above analysis be helpful to solve our main problem. Well, we would like to achieve the bound mentioned in Equation 1 on the collisions for a given fixed set S. Here is a brilliant idea :

Can we find a family of reasonably good *hash functions ? If so, then we may select a random function from there and try to achieve bound stated in Equation 1.*

In this manner, we essentially are shifting the randomization from the distribution of set S over U to the distribution of the hash function over a given hash family.

Now how to formalize the goodness of a hash family ? If you have understood the above analysis for a randomly selected S, you would realize that what is needed is that the probability that a pair of elements from U collide is less than c/n for some constant. This introduces the idea of universal hash family as follows.

2.2 Universal Hash family

Definition 2.1 *A hash family H is called good if for any pair $a, b \in U$ and a randomly selected function $h \in H$, probability that $h(a) = h(b)$ is less than c/n for some constant c.*

$$\Pr_{h \in_r H}[h(a) = h(b)] \leq \frac{c}{n}$$

Though the family of all functions from U to $\{0, ..., n-1\}$ satisfies the definition of universal hash family, there are some serious concerns about using this *huge* family (Ponder over this point). So what is needed is a c-universal hash family which is quite *small* as well. In fact there are many such universal hash families. From now onwards, we shall pursue two aims which can be studied quite independent of each other. The first one is to use a given c-universal hash family to find out a perfect hash function for a given set S. The second aim is to find out a *small* universal hash family.

3 Perfect hashing using a universal hash family

We shall first devise a hashing scheme for S which uses $\Theta(s^2)$ space. We shall then use some simple and nice ideas to reduce the space to $O(s)$.

3.1 Perfect hashing with $n = \Theta(s^2)$ space

Let H be a given universal hash family. For set S, let us calculate $\mathbf{E}[Y]$, the expected number of collision. Let us introduce random variable Y_{ab} for any $a, b \in S$ as follows.

$$Y_{ab}$$

Exploiting linearity of expectation, and the definition of universal hashing, it follows that

$$\begin{aligned}
\mathbf{E}[Y] &= \sum_{a,b \in S, a<b} \mathbf{E}[Y_{ab}] \\
&= \sum_{a,b \in S, a<b} \Pr_{h \in_r H}[h(a) = h(b)] \\
&\leq \sum_{a,b \in S, a<b} \frac{c}{n} = \frac{s(s-1)}{2}\frac{c}{n} \leq \frac{cs^2}{2n}
\end{aligned}$$

Ponder over the following question before proceeding further.

Question 3.1 *How to use the above equation to get a perfect hashing ?*

It follows from the above analysis that *larger* the hash table, fewer will be the number of collisions, in expectation. This hints suggests that in order to get a perfect hash function, we should have a sufficiently large n. The above equation and Markov inequality imply the following bound (it is also a lesson for those who underestimate the power of Markov Inequality).

$$\Pr[Y \neq 0] = \Pr[Y \geq 1] \leq \frac{\mathbf{E}[Y]}{1} = \frac{cs^2}{2n}$$

Hence if we choose $n = cs^2$, then a random hash function from H is going to be a perfect hash function for S with probability at least $1/2$. So here is an algorithm for perfect hashing with space $O(s^2)$: Build a table of size $n > cs^2$. Select a random hash function from H. Hash all elements of S using this function, and repeat if there is any collision. It follows that expected number of repetitions to find a perfect hash function is $O(1)$. So we can conclude the following theorem.

Theorem 3.1 *A set S can be preprocessed in expected (s^2) time to build a hash table of $O(s^2)$ size which can support any search query in worst case $O(1)$ time.*

3.2 Perfect hashing with $n = O(s)$

Our objective is to compute a hashing for S which uses only $O(s)$ space. It follows that for $n = O(s)$, if we select a hash function randomly uniformly from H, then it is unlikely to be a perfect hash function. So what should we do ? We should have patience and investigate further instead of giving up.

Though for $n = s$, a randomly picked hash function is not likely to be perfect, it still makes sense as to how bad it could be. So we pick a uniformly random hash function from H, build the hash table of size $n = O(s)$, and store all the elements of set S into this hash table. There may be multiple elements from S which will be hashed to the same index in T. Let Z_i denote the random variable for the number of all those elements from S which are mapped to i. In order to determine how bad a uniformly random hash function $h \in H$ will prove to be, we enquire about the expected number of collisions for $n = \Theta(s)$. It follows from the discussion above that

$$\mathbf{E}[Y] \leq \frac{cs^2}{2n} = \frac{cs}{2} \quad \text{for } n = s \tag{2}$$

So we can state the following lemma (it is again a simple application of Markov Inequality).

Lemma 3.1 *If $n = s$, then the total number of collisions Y will exceed cs with probability at most $1/2$.*

Note that the worst possible number of collision for a set of s elements is $\binom{s}{2} = \Theta(s^2)$. So Lemma 3.1 conveys that even for $n = s$, the universal hash family ensures very few collisions, in expectation. So is there some way to use this lemma. Can you think of exploiting Theorem 3.1 somehow ? Ponder over it before proceeding further.

Since the number of collisions will be few, let us do the following. For each index i, if $Z_i > 1$, build a hash table of size $O(Z_i^2)$, as stated in Theorem 3.1), for elements hashed to index i. Let h_i denote the corresponding hash function. This gives a data structure with 2 levels of hashing. It will surely give $O(1)$ worst case search time as follows. For any element x, first compute the first level hash function $h(x)$ to know the index i in the first level hash table, and then use $h_i(x)$ to look for x in second level hash table.

What about the overall size of this data structure ? The primary hash table uses $O(s)$ space. For all the second level hash tables, the total size is of the order of $\sum_i Z_i^2$. Is there some way to calculate $\mathbf{E}[\sum_i Z_i^2]$? Can we the random variable Y (the number of collisions) in terms of Z_i's. Ponder over this question before proceeding further.

For ith index, the total number of collisions will be $\binom{Z_i}{2}$. Combining over all indices $0 \leq i \leq n-1$, we get

$$\sum_{0 \leq i \leq n-1} \binom{Z_i}{2} = Y$$

Expanding each term of LHS and rearranging the terms, we get

$$\sum_{0 \leq i \leq n-1} Z_i^2 = 2Y + \sum_{0 \leq i \leq n-1} Z_i = 2Y + s \tag{3}$$

Combining Equations 2 and 3, we get $\mathbf{E}[\sum_i Z_i^2] \leq (c+1)s$. Great !! This is what we dreamt to achieve.

Algorithm 1 describes the steps for computing a hashing for S with worst case $O(1)$ search time. Using Theorem 3.1 and simple probabilistic arguments, it is easy to observe that the expected time required to

Algorithm 1: Computing a perfect hashing in $O(s)$ space using a universal hash family$H$

```
flag ← FALSE;
while flag=FALSE do
    h ←_r H;
    Hash all the elements from S to the hash table T according to h;
    Y ← number of collisions in S under h;
    if Y is less than O(s) then flag ← TRUE
for each index i with Z_i > 1 do
    compute a perfect hash family with space O(Z_i^2) as mentioned in Theorem 3.1.
```

compute this hashing scheme is linear in s. As usual, we have omitted the time complexity of selecting a function randomly uniformly from H.

Theorem 3.2 *A given set $S \subseteq U$ can be preprocessed in expected $O(s)$ time to build a data structure of size $O(s)$ which takes $O(1)$ worst case time to determine whether or not any elment $x \in U$ is there in S.*

4 Example of a universal hash family

As mentioned above, the set of all possible functions from U to $\{0, ..., n-1\}$ is a universal hash family. We shall now describe a 2-universal hash family which is too *small*. A hash function selected from this family will require $O(\log p)$ bits which is $O(1)$ words. This universal hash family is derived from the earlier hash function ($h(a) = a \bmod n$) with a very natural intuition and elementary knowledge of prime numbers. Here it is :

Let p be a prime number in the interval $[m, 2m]$. For any $1 \leq a \leq p-1$, let us define function $h_a(x) = ax \bmod p \bmod n$. Consider the hash family

$$H = \{h_a \mid 1 \leq a \leq p-1\}$$

Make sincere attempts on your own to prove that H is 2-universal.

Points to reflect upon :

1. How crucial is the role of prime numbers in the universal hashing scheme described above ?
2. Try to find out the key ideas and tools used to arrive at perfect hashing scheme with linear space. In particular, explore the importance of universal hash family and convince yourself that its underlying idea plays the key role in achieving perfect hashing.
3. Compare the universal hash family $H = \{h_a | 1 \leq a \leq p-1\}$ with the set of all functions from U to $\{0, ..., n-1\}$. What are the advantages and disadvantages of one on the other ?
4. There are other ways to construct universal hash families which are small. Think about devising them.
5. In this lecture we devised a static hashing scheme since S was static. In real life S may be dynamic. There are many dynamic hashing schemes as well. One of them is "cuckoo hashing". Motivated students should study it. It is available at `http://www.itu.dk/people/pagh/papers/cuckoo-jour.pdf`