CS787: Advanced Algorithms

Scribe: Mayank Maheshwari, Chris Hinrichs **Lecturer:** Shuchi Chawla
Topic: Hashing and NP-Completeness **Date:** September 21 2007

Previously we looked at applications of randomized algorithms, and began to cover randomized hashes, deriving several results which will be used in today's lecture notes. Today we will finish covering Randomized hashing and move on to the definition of the classes P and NP, and NP-Completeness.

8.1 Hashing Algorithms

Sometimes it is desireable to store a set of items S taken from a universe U using an amount of storage space n and having average lookup time $O(1)$. One system which has these properties is a hash table with a completely random hash function. Last time we saw how to implement this using a 2-universal hash family. Unfortunately these hash families can have a large lookup time in the worst case. Sometimes we want the worst case lookup time to also be $O(1)$. We will now see how to achieve this *perfect hashing*.

8.1.1 Perfect hashing – reducing worst case loopkup time

Given a 2-uniform hashing function h (informally, a function which has a $1/n$ probability of assigning 2 different independently chosen elements of U to the same value i, also called "bin i") and that $|S| = m$, $|U| = u$ and $S \subseteq U$ we can construct an array $[n]$ of n bits such that if an element $i \in S$ maps to $h(i)$ then the array answers that i is in S. Note that there is a certain probability that this will be false because elements not in S can be mapped to the same location by the hash function. If there are more than one element in S mapped to the same bit i then the single bit in the array is replaced by a linked list of elements, which can be searched until the desired element is found. Under this arrangement, the worst case lookup time is determined by the longest expected linked list in $[n]$, which is equivalent to the expected number of assignments of elements in S to a single bin, i.e. collisions in bin i. The expectation of the number of collisions at bin i is given by

$$\mathbf{E}[X_i] = 1/n \binom{m}{2} \tag{8.1.1}$$

where X_i is a random variable representing the number of elements in S mapped to bin i. In order for the expected lookup time to be $O(1)$ the quantity on the right hand side of (8.1.1) must be $O(1)$ which means that n must be $O(m^2)$.

In order to ensure that $\mathbf{E}[X_i] = O(1)$ we can repeatedly choose new hash functions h from a 2-Universal family of functions $H : U \to [n^2]$ until one is found which produces no more than $O(1)$ collisions in any bin.

8.1.2 Perfect hashing with linear space

An improvement on this method of hashing elements of S which reduces its space complexity is to choose $n \approx m$ and instead of using a linked list of elements which map to the same bin, we can use another hash of size n to store the elements in the list. Let b_i be the number of elements in bin i. We note that the expected maximum number of elements in a single bin, is roughly $\sqrt{n}$ because we know from theorem 7.1.1 that $\mathbf{Pr}[\max_i b_i \geq k] \approx 1/k^2$ and thus $\mathbf{Pr}[\text{any } b_i \geq \sqrt{n}] \approx 1/n$, so we accpet with high confidence that no bin has more than $\sqrt{n}$ items in it. The size of this hash is roughly the square of the number of elements to be stored in it, making its expected lookup time $O(1)$ as discussed above.

If b_i is the number of elements placed in bin i, i.e. the number elements in the sub-hash in bin i, (if $b_i \neq 1$), then the total amount of space used by this method, T_{sp} is given by:

$$T_{sp} = m + \mathbf{E}\left[\sum_i b_i^2\right] \tag{8.1.2}$$

The first term is for the array itself, and the second is for the sub-hashes, each of which stores b_i items, and requires b_i^2 space. We can note that

$$\mathbf{E}\left[\sum_i b_i^2\right] = \sum_i 2 * \binom{b_i}{2} + \sum_i b_i \tag{8.1.3}$$

because

$$\binom{b_i}{2} = \frac{b_i * (b_i - 1)}{2}$$

The $\sum_i b_i$ term is equal to m, the number of items stored in the main hash array. We can approximate the summation term my saying that

$$\sum_i \binom{b_i}{2} \approx \frac{1}{m}\binom{m}{2}$$

T_{sp} then becomes

$$T_{sp} \approx 2m + \frac{2}{m}\frac{m(m-1)}{2} \approx 3m \tag{8.1.4}$$

8.2 Bloom Filters

Before we go on to the idea of bloom filters, let us define the concept of *false positive.*

Definition 8.2.1 (False Positive) *A given element $e \in U$ and $|S| = m$ and the elements of S are mapped to an array of size n. If the element $e \notin S$ but the hashing algorithm returns a 1 or "yes" as an answer i.e. the element e is mapped to the array, it is called a false positive.*

Now the probability of a false positive using perfect hashing is:

$$\begin{aligned} \mathbf{Pr}[\text{a false positive}] &= 1 - \mathbf{Pr}[0 \text{ in that position}] \\ &= 1 - (1 - \frac{1}{n})^m \\ (n \gg m) & \\ &\approx 1 - (1 - \frac{m}{n}) = \frac{m}{n} \end{aligned}$$

This probability of getting a false positive is considerably high. So how can we decrease this?

By using 2-hash functions, we can reduce this probability to $(\frac{m}{n})^2$. So, in general, by using k-hash functions, we can decrease the probability of getting a false postive to a significantly low value. So, a fail-safe mechanism to get the hashing to return a correct value by introducing redundancy in the form of k-hash functions is the basic idea of a bloom filter. *Problem:*Given an $e \in U$, we have to map S to $[n]$ using k hash functions.

So to solve this problem, we

- Find $h_1(e), h_2(e), \ldots, h_k(e)$.
- If any of them is 0, output $e \notin S$, else output $e \in S$.

Now the $\mathbf{Pr}$[a false positive] $= 1 - \mathbf{Pr}$[any one position is 0].

So let's say the

$$\mathbf{Pr}[\text{some given position is 0}] = (1 - \frac{1}{n})^{mk} \approx \exp(\frac{-mk}{n}) = (\text{say})p. \tag{8.2.5}$$

So $\mathbf{Pr}$[any given position is 1]$= 1-p$. And $\mathbf{Pr}$[all positions that e maps to are 1 given $e \notin S$]$=(1-p)^k$.

This is correct if all the hash functions are independent and k is small.

*Problem:*The probability of getting a false positive using k-hash functions is $(1-p)^k$. So how can we minimize it?

Let us say the function to be minimized is $f(k) = (1 - \exp(\frac{-km}{n}))^k$.

By first taking the log of both sides and then finding the first derivative, we get

$$\begin{aligned}\frac{d}{dk}\log(f(k)) &= \frac{d}{dk}k\log(1-\exp(\frac{-km}{n})) \\ &= \log(1-\exp(\frac{-km}{n})) + \frac{k}{1-\exp(\frac{-km}{n})}\exp(\frac{-km}{n})\frac{m}{n} \\ &= \log(1-p) - \frac{p}{1-p}\log(p)\end{aligned}$$

By substituting $p = \exp(\frac{-km}{n})$ and $\frac{km}{n} = -\log(p)$

To minimize this function, we need to solve the equation with its first derivative being equal to 0.

So to solve this equation:

$$\begin{aligned}&\log(1-p) - \frac{p}{1-p}\log p = 0 \\ \Rightarrow\ &(1-p)\log(1-p) = p\log p\end{aligned}$$

Solving this equation gives $p = \frac{1}{2}$.

So we get

$$\exp(\frac{-km}{n}) = \frac{1}{2} \Rightarrow k = \frac{n}{m}\ln 2$$

So with that value of k, we can deduce the value of $f(k)$ as:

$$\begin{aligned}f(k) &= \mathbf{Pr}[\text{positive}] \\ &= (1-p)^k \\ &= \frac{1}{2}^{\frac{n}{m}\ln 2} \\ &= c^{\frac{n}{m}}.\end{aligned}$$

If we choose $n = m\log m$ where $c < 1$ is a constant, this gives $f = \frac{1}{m}$.

So this way, we can reduce the probability of getting a false positive by using k-hash functions.

8.3 NP-Completeness

8.3.1 P and NP

When analyzing the complexity of algorithms, it is often useful to recast the problem into a decision problem. By doing so, the problem can be thought of as a problem of verifying the membership of a given string in a language, rather than the problem of generating strings in a language. The

complexity classes P and NP differ on whether a witness is given along with the string to be verified. P is the class of algorithms which terminate in an amount of time which is $O(n)$ where n is the size of the input to the algorithm, while NP is the class of algorithms which will terminate in an amount of time which is $O(n)$ if given a witness w which corresponds to the solution being verified. More formally,

$L \in$ NP iff $\exists$ P-time verifier V $\in$ P s.t.

$\forall$ x $\in$ L, $\exists$ w, $|w| = \text{poly}(|x|)$, V(x, w) accepts

$\forall$ x $\notin$ L, $\forall$ w, $|w| = \text{poly}(|x|)$, V(x, w) rejects

The class Co-NP is defined similarly:

$L \in$ Co-NP iff $\exists$ P-time verifier V $\in$ P s.t.

$\forall$ x $\notin$ L, $\exists$ w, $|w| = \text{poly}(|x|)$, V(x, w) accepts

$\forall$ x $\in$ L, $\forall$ w, $|w| = \text{poly}(|x|)$, V(x, w) rejects

An example of a problem which is in the class NP is Vertex Cover. The problem states, given a graph $G = (V, E)$ find a set $S \subseteq V$, then $\forall e \in E$, e is incident on a vertex in S, and $|S| \leq k$ for some number k. There exists a verifier V by construction; V takes as input a graph G, and as a witness a set $S \subseteq V$ and verifies that all edges $e \in E$ are incident on at least one vertex in S and that $|S| \leq k$. If G has no vertex cover of size less than or equal to k then there is no witness w which can be given to the verifier which will make it accept.

8.3.2 P-time reducibility

There exist some problems which can be used to solve other problems, as long as a way of solving them exists, and a way of converting instances of other problems into instances of the problem with a known solution also exists. When talking about decision problems, a problem A is said to reduce to problem B if there exists an algorithm which takes as input an instance of problem A, and outputs an instance of problem B which is guaranteed to have the same result as the instance in problem A, i.e. if L_A is the language of problem A, and L_B is the language of problem B, and if there is an algorithm which translates all $l \in L_A$ into $l_B \in L_B$ and which translates all $l' \notin L_A$ into $l'_B \notin L_B$ then problem A reduces to problem B.

The practical implication of this is that if an efficient algorithm exists for problem B, then problem A can be solved by converting instances of problem A into instances of problem B, and applying the efficient solver to them. However, the process of translating instances between problems must also be efficient, or the benefit of doing so is lost. We therefore define P-time reducibility as the process of translating instances of problem A into instances of problem B in time bounded by a polynomial in the size of the instance to be translated, written $A \leq_P B$. This reduction is also called Cook reduction.

8.3.3 NP-Completeness

If it is possible to translate instances of one problem into instances of another problem, then if there exists a problem L such that

$$\forall L' \in \text{NP},\ L' \leq_P L$$

then an algorithm which decides L decides every problem in NP. Such problems are called NP-Hard. If a problem is in NP-Hard and NP, then it is called NP-Complete. If there exists a P-time algorithm which decides an NP-Complete problem, then all NP problems can be solved in P-time, which would mean that $NP \subseteq P$. We already know that $P \subseteq NP$ because every P-time algoritm can be thought of as an NP algorithm which takes a 0-length witness. Therefore, $P = NP$ iff there exists a P-time algorithm which decides any NP-Complete problem. This result was proved independently by Cook and Levin, and is called the Cook-Levin theorem. The first problem proved to be in NP-Complete was Boolean SAT, which asks, given a Boolean expression, is there a setting of variables which allows the entire expression to evaluate to True?