U.C. Berkeley – CS170 : Algorithms
Lecturers: Sanjam Garg and Prasad Raghavendra April 25, 2017

Sublinear Algorithms

In recent years, there is a growing need to design algorithms that can handle inputs of astronomical sizes. This calls for algorithms whose space/time requirements are sub-linear in the input size, i.e., much less than $O(n)$, say a small polynomial in $\log n$. We will see a few examples of sub-linear algorithms in this lecture.

1 Sampling

Perhaps, the simplest and most fundamental sub-linear time algorithm is *random sampling.* To illustrate the algorithm, let us consider a toy example.

Suppose we have a population of 300 voters participating in an election with two parties A and B. Our goal is to determine the fraction of the population voting for A. A simple sub-linear time algorithm would be to sample t voters out of 300 uniformly at random, and compute the fraction of the sampled voters who vote for A.

Let p denote the fraction of the population voting for A. Suppose we sample t people, each one independently and uniformly at random from the population. Let X_i denote the 0/1 random variable indicating whether the i^{th} sample, votes for A. The algorithm's estimate for p is given by,

$$\tilde{p} = \frac{1}{t}\sum_{i=1}^{t} X_i \,.$$

It is easy to check that,

$$\begin{aligned}\mathbb{E}[X_i] &= \mathbb{P}[X_i = 1]\cdot 1 + \mathbb{P}[X_i = 0]\cdot 0\\ &= p\cdot 1 + (1-p)\cdot 0 = p\,,\end{aligned}$$

and therefore $\mathbb{E}[\tilde{p}] = \frac{1}{t}\sum_i \mathbb{E}[X_i] = p$. So the expected value of the estimate $\tilde{p}$ is exactly equal to p (such an estimator is said to be *unbiased*).

The natural question to answer is, how many samples do we need in order to ensure that the estimate $\tilde{p}$ is within say 0.1 of the correct value p, with probability say 0.9. To answer this question, we will appeal to the following theorem.

Theorem 1.1. *(Chernoff/Hoeffding Bound) Suppose $X_1 \ldots, X_t$ are i.i.d random variables taking values in $\{0,1\}$. Let $p = \mathbb{E}[X_i]$ and $\varepsilon > 0$, then*

$$\mathbb{P}\left[\left|\frac{1}{t}\cdot\sum_{i=1}^{t} X_i - p\right| \geq \varepsilon\right] \leq 2e^{-2\varepsilon^2 t}$$

In particular, if we want to ensure that,

$$\mathbb{P}[\text{Estimate has an error } \geq \varepsilon] \leq \delta$$

then we need to set $t = \lceil \frac{1}{2\varepsilon^2}\log_e\left(\frac{2}{\delta}\right)\rceil$.

Notice that the number of samples needed to ensure that the estimate is within error ε with probability $1-\delta$, is independent of the population size! In other words, to obtain a 0.1-approximate estimate with probability 0.9, we need to sample the same number of voters irrespective of whether the total population is 300 or 300 million. In this sense, the running time of random sampling algorithm is not just sub-linear in input, but independent of the input size!

2 Counting Distinct Elements

Consider the following toy problem: you are given a sequence of words $w_1, \ldots, w_n$ (say a really large piece of literature) and the goal is to estimate the number of distinct words in the sequence.

A trivial solution would involve scanning the words $w_1, \ldots, w_n$ once, while remembering at each point in time, all the words seen. In general, this algorithm requires $\Omega(n)$-bits of memory. Now, we will see an algorithm that uses sub-linear space, in fact, $poly(\log n)$ space.

Streaming Model. The above toy problem arises in various guises in many applications. For example, consider a network router monitoring internet traffic that it is forwarding. Suppose at the end of the a day, the network administrator would like to estimate the number of different IP addresses seen by the router.

Here the algorithm has to face two severe restrictions:

- The space available at the router is too little to store all the IP addresses seen in the day.
- The router sees the stream of traffic (the input) only once, i.e., the algorithm can read the stream only once.

An algorithm designed with these restrictions is known as a *streaming algorithm*. Specifically, in the streaming model, the input is a stream of symbols $x_1, \ldots, x_n$ from some domain, say $\{1, \ldots, N\}$. The goal is to compute some function $f(x_1, \ldots, x_n)$. However, the space available to the algorithm is poly$(\log n)$ and the algorithm reads the input stream exactly once in the order $x_1, \ldots, x_n$.

Distinct Elements Algorithm. We are now ready to describe a streaming algorithm for counting the number of distinct words. Let W denote the set of all possible words. First, we will describe an idealized algorithm to illustrate the main idea.

```
1: Pick a hash function $h : W \to [0,1]$
2: $currentmin \leftarrow 1$
3: for $i = 1$ to $n$ do
4:    if $h(w_i) < currentmin$ then
5:       $currentmin \leftarrow h(w_i)$
6:    end if
7: end for
8: Output $\frac{1}{currentmin}$
```

Algorithm 1: Counting Distinct Elements

The algorithm finds the minimum hash value of a word in the stream, and outputs its inverse. It is easy to check that the algorithm can be implemented with $O(\log n)$ bits of space (after suitably discretizing the hash function).

To describe the main idea behind the algorithm, let us first make a strong assumption.

(Random Hash Assumption) *For each word $w \in W$, its hash value $h(w)$ is a uniformly random number in $[0,1]$ independent of all other hash values*

Suppose the stream $w_1, \dots, w_n$ contains k different words, then the algorithm encounters k different hash values. If $r_1, \dots, r_k$ are these k-different hash values, then the algorithm will output $\frac{1}{\min(r_1,\dots,r_k)}$.

If $r_1, \dots r_k$ are k independently chosen random numbers in $[0,1]$ then we expect these numbers to be uniformly distributed in $[0,1]$ and the smallest of them to be around $\frac{1}{k}$. Therefore, we can expect that the reciprocal of the minimum is approximately k – the number of distinct words.

More formally, one can show that

Lemma 2.1. *Suppose $r_1, \dots, r_k$ are independently and uniformly distributed in $[0,1]$ then,*

$$\mathbb{E}[\min(r_1, \dots, r_k)] = \frac{1}{k+1}$$

Proof.

$$\begin{aligned}
\mathbb{E}[\min(r_1, \dots, r_k)] &= \int_{r_1,\dots,r_k} \min(r_1, \dots, r_k) dr_1 dr_2 \dots dr_k \\
&= \int_{r_1,\dots,r_k} \left(\int_{r_{k+1}=0}^{1} 1[r_{k+1} \le \min(r_1, \dots, r_k)] dr_{k+1} \right) dr_1 dr_2 \dots dr_k \\
&= \mathop{\mathbb{P}}_{r_1,\dots,r_k,r_{k+1}} [r_{k+1} \le \min(r_1, \dots, r_k)]
\end{aligned}$$

where $r_1, \dots, r_k, r_{k+1}$ are uniformly random elements in $[0,1]$. But, $r_{k+1} \le \min(r_1, \dots, r_k)$ if and only if $r_{k+1} = \min(r_1, \dots, r_{k+1})$. The claim follows by observing that $r_{k+1} = \min(r_1, \dots, r_{k+1})$ with probability exactly $\frac{1}{k+1}$, since any of the $k+1$ elements $\{r_1, \dots, r_{k+1}\}$ is equally likely to be the smallest element. □

While the above analysis captures the basic intuition behind the algorithm, there is a major flaw in the argument. Specifically, the analysis crucially relied on the assumption that the hash values of different words are independent uniformly distributed random values in $[0,1]$.

This assumption is too strong to hold for any reasonable hash function family. In fact, if a hash function $h : W \to [0,1]$ were truly random, then the only way to represent the hash function would be by its truth table which would require at least $|W|$ bits of memory.

For the streaming algorithm to be space-efficient, we will need to pick hash functions that have a small representation, say using only poly$(\log n)$ bits of memory. To this end, we will use a hash function chosen from a universal hash family.

Definition 2.2. (Universal Hash Family)

A family of functions $\mathcal{H} = \{h_1, \dots, h_M\}$ from a domain D to a range R, is said to be a universal hash family if the following holds: If we pick a hash function h at random from $\mathcal{H}$, then on any pair of inputs $x, y \in D$, the behaviour of h exactly mimics that of a completely random function. Formally, for all $x \neq y \in D$ and $i, j \in R$,

$$\mathop{\mathbb{P}}_{h \in \mathcal{H}} [h(x) = i \wedge h(y) = j] = \frac{1}{|R|^2}$$

Notice that a hash function chosen from a universal hash family looks random if we only look at two inputs. If we consider three different hashes $h(x), h(y)$ and $h(z)$ simultaneously, then the three values might not appear random at all.

A simple and elegant example of a universal hash family is as follows. Fix a prime number p. For each $a \in Z_p$ and $b \in \mathbb{Z}_p$, define $h_{a,b} : \mathbb{Z}_p \to \mathbb{Z}_p$ as,

$$h_{a,b}(x) = a \cdot x + b \mod p\,,$$

and the hash family $\mathcal{H} = \{h_{a,b} | a \in Z_p, b \in \mathbb{Z}_p\}$. $\mathcal{H}$ is a universal hash family such that any function in the family can be represented $O(\log p)$ bits by storing two integers $a, b \in Z_p$.

Yet another finer implementation detail is that the output range of the hash function cannot be the continous domain $[0, 1]$, but would need to be discretized. In the correct implementation of the counting distinct elements algorithm, the algorithm discretizes $[0, 1]$ in to p values, and picks the hash function h to be from a universal hash family.

The formal analysis of the algorithm is more involved (check out http://inst.eecs.berkeley.edu/~cs170/fa16/lecture-11-22.pdf if interested).

3 Detecting Duplicates

Consider the following problem: suppose we have n documents $D_1, \ldots, D_n$ and would like to detect duplicates amongst them.

The naive algorithm would compare every pair of documents with each other which can be very impractical. An efficient solution to this problem of detecting duplicates would be the following:

- Pick a hash function $H : \{Documents\} \to \{1, \ldots, n\}$.
- Apply the hash function to each document and only compare documents with the same hash value.

While the above algorithm is efficient, it has one major drawback. The algorithm is only useful towards detecting documents that are exact copies of each other. Specifically, if two documents D_1, D_2 are nearly the same, except for a minor change (say a deleted word), then typically $H(D_1) \neq H(D_2)$. Therefore the above algorithm does not detect the pair of nearly identical documents D_1 and D_2.

In many applications, the goal is to detect all pairs of documents that are very similar – say, share 90% of text. For these applications, we need a *similarity preserving* hash function H such that,

$$D_1 \text{ is similar to } D_2 \implies H(D_1) \approx H(D_2)\,.$$

Now, we will see how to construct such a hash function H on documents. The idea is to pick a hash function $h : \{Words\} \to [0, 1]$, and define H as,

$$\begin{aligned} H(\text{Document} D) &= \text{the word in document } D \text{ with smallest hash value} \\ &= \operatorname*{arg\,min}_{\text{word } w \text{ in } D} h(w) \end{aligned}$$

The following lemma shows that H is a similarity preserving hash function.

Lemma 3.1. *For any pair of documents D_1, D_2,*

$$\mathbb{P}_h[H(D_1) = H(D_2)] = \frac{\#\ \textit{words common to } D_1 \textit{ and } D_2}{\#\textit{words in } D_1 \cup D_2}$$

The quantity on the right hand side is known as *Jaccard similarity measure* between documents. If D_1 and D_2 are identical then it is equal to 1, and if D_1 and D_2 don't share common words then it is equal to 0. For every pair of documents, the Jaccard similarity measure is a number in $[0, 1]$.

Proof. (Proof of Lemma 3.1)

Let w^* be the word with smallest hash value in $D_1 \cup D_2$. First, we claim that $H(D_1) = H(D_2)$ if and only if $w^* \in D_1 \cap D_2$.

Let us assume this claim, and see how the lemma follows. By the claim,

$$\mathbb{P}[H(D_1) = H(D_2)] = \mathbb{P}[w^* \in D_1 \cap D_2]\,.$$

Assuming the hash values of words are independently and uniformly distributed in $[0, 1]$, every word in $D_1 \cup D_2$ is equally likely to be the one with the smallest hash value. Therefore, we get that

$$\mathbb{P}[w^* \in D_1 \cap D_2] = \frac{|D_1 \cap D_2|}{|D_1 \cup D_2|}\,.$$

The above two equations imply the lemma.

Now, let us see why the claim is true. Suppose $H(D_1) = H(D_2) = w$ for a word w. By definition of the hash function H, $h(w)$ is the smallest hash value of any word in D_1, and $h(w)$ is the smallest hash value of any word in D_2. Therefore the word w has the smallest hash value of any word in $D_1 \cup D_2$, i.e., $w^* = w$. Also by definition of H, $w = H(D_1) \in D_1$ and $w = H(D_2) \in D_2$, implying that $w^* \in D_1 \cap D_2$ as needed.

Conversely, if $w^* \in D_1 \cap D_2$ it is easy to see that $H(D_1) = H(D_2) = w^*$. By definition, w^* is the word with smallest hash value in $D_1 \cup D_2$. If $w^* \in D_1 \cap D_2$ then w^* is also the word with smallest hash value within the sets D_1 and D_2 too, i.e., $H(D_1) = H(D_2) = w^*$.

□