CS 170 DIS 13

Released on 2019-04-23

1 Document Comparison with Streams

You are given a document A and then a document B, both as streams of words. Find a streaming algorithm that returns the degree of similarity between the words in the documents, given by $\frac{|I|}{|U|}$, where I is the set of words that occur in both A and B, and U is the set of words that occur in at least one of A and B.

Clearly explain your algorithm and briefly justify its correctness and memory usage (at most $\log(|A| + |B|)$. Can we achieve accuracy to an arbitrary degree of precision? That is, given any $\epsilon > 0$ can we guarantee that the solution will always be within a factor of $1 \pm \epsilon$ with high probability?

Solution: Simply use the number of distinct elements streaming algorithm we saw in class, on the streams of A, B, and $C := A \cup B$ (for this third stream, process words in A and words in B). Let $|A|, |B|$, and $|C|$ be the output of the algorithm on the corresponding set, our estimate for the number of distinct words in it. Then, $|U| = |C|$, and $|I| = |A| + |B| - |U|$. Thus our estimate for $|I|/|U| = (|A| + |B| - |C|)/|C|$.

Memory usage is logarithmic in the length of the documents, as we're using a constant number (three) copies of the streaming algorithm shown in class, which used logarithmic memory. Correctness flows from the correctness of the underlying streaming algorithm for distinct elements, combined with the elementary axiom in set theory that $|A \cap B| = |A| + |B| - |A \cup B|$.

We can achieve accuracy within an arbitrary factor, because the accuracy of the similarity estimate depends on the accuracy of the underlying number of distinct elements algorithm, which we saw in class gives a result which is with high probability a fraction $(1 \pm \epsilon)$ of the true value.

The full mathematical details are beyond our scope, but we can do a rough analysis to gauge our algorithm's accuracy: the numerator is the sum of three outputs of the streaming algorithm, so is with high probability $1 \pm 3\epsilon$, and the denominator is within $1 \pm \epsilon$, of their true values, so the overall fraction is with high probability within $(1 \pm 3\epsilon)/(1 \mp 1\epsilon)$. The worst case in this range would be $(1 + 3\epsilon)/(1 - \epsilon)$, (or swap the + and - signs), which simplifies to $\frac{(1+3\epsilon)(1+\epsilon)}{(1-\epsilon)(1+\epsilon)} = \frac{1+4\epsilon+3\epsilon^2}{1-\epsilon^2} \approx 1 + 4\epsilon$, because ϵ^2 is negligible. In the other case, we obtain $1 - 4\epsilon$, so we conclude the overall accuracy is likely to be within $1 \pm 4\epsilon$. This calculation is not exact, but suffices to argue that the accuracy of the overall algorithm is a function of the accuracy of the underlying number of distinct elements algorithm it used, which can be set arbitrarily low.

2 Lower Bounds for Streaming

(a) Consider the following simple 'sketching' problem. Preprocess a sequence of bits $b_1, \ldots, b_n$ so that, given an integer i, we can return b_i. How many bits of memory are required to solve this problem exactly?

(b) Given a stream of integers $x_1, x_2, \ldots$, the *majority element* problem is to output the

integer which appears most frequently of all of the integers seen so far. Prove that any algorithm which solves the majority element problem exactly must use $\Omega(n)$ bits of memory, where n is the number of elements seen so far.

Solution:

(a) n bits. Intuitively, this is because at the end of the preprocessing, there are 2^n different 'states' the algorithm has to be in, one for each bitstring. The number of states of a machine with ℓ bits of memory is 2^ℓ, so $\ell \geq n$. A more detailed argument follows.

The preprocessing algorithm is a function $f\colon \{0,1\}^n \to \{0,1\}^\ell$, where ℓ is the number of bits of memory needed to answer queries. The query algorithm is a function $q\colon [n] \times \{0,1\}^\ell \to \{0,1\}$. Observe that we can use the query algorithm to invert f on its image: if $g(y) = (q(1,y), q(2,y), \ldots, q(n,y))$, then $g(f(x)) = x$ for all $x \in \{0,1\}^n$. Hence f is injective, which means that $\ell \geq n$.

(b) We can prove this by reduction from the previous problem. For any string of bits $b_1, \ldots, b_\ell$, we define a stream of integers $0, 0, (i,i)_{b_i=1}$. Now we can query b_i by adding (i) to the stream and checking if it is the majority element. The length of the sequence is $n \leq 2\ell + 1$, so the memory usage is at least $\frac{n-1}{2}$ bits.

An alternative approach is for the stream to be $((-1)^{b_i} \cdot i)_{i \in [n]}$. Then we can query b_i by adding $(i, -i)$ to the stream. If $-i$ is the majority element, then $b_i = 1$, otherwise $b_i = 0$.

3 Universal Hashing

Let $[m]$ denote the set $\{0, 1, ..., m-1\}$. For each of the following families of hash functions, determine whether or not it is universal. If it is universal, determine how many random bits are needed to choose a function from the family.

(a) $H = \{h_{a_1,a_2} : a_1, a_2 \in [m]\}$, where m is a fixed prime and

$$h_{a_1,a_2}(x_1, x_2) = a_1 x_1 + a_2 x_2 \mod m$$

Notice that each of these functions has signature $h_{a_1,a_2} : [m]^2 \to [m]$, that is, it maps a pair of integers in $[m]$ to a single integer in $[m]$.

(b) H is as before, except that now $m = 2^k$ is some fixed power of 2.

(c) H is the set of all functions $f : [m] \to [m-1]$.

Solution:

(a) The hash function is universal. The universality proof is the same as the one in the textbook (only now we have a 2-universal family instead of 4-universal). To reiterate, assume we are given two distinct pairs of integers $x = (x_1, x_2)$ and $y = (y_1, y_2)$. Without loss of generality, let's assume that $x_1 \neq y_1$. If we chose values a_1 and a_2 that hash x and y to the same value, then $a_1 x_1 + a_2 x_2 \equiv a_1 y_1 + a_2 y_2 \mod m$. We can rewrite this as $a_1(x_1 - y_1) \equiv a_2(y_2 - x_2) \mod m$. Let $c \equiv a_2(y_2 - x_2) \mod m$. Since m is prime and

$x_1 \neq y_1$, $(x_1 - y_1)$ must have a unique inverse. So $a_1(x_1 - y_1) \equiv a_2(y_2 - x_2) \mod m$ if and only if $a_1 \equiv c(x_1 - y_1)^{-1} \mod m$, which will only happen with probability $1/m$.

We need to randomly pick two integers in the range $[0, \ldots, m-1]$, so we need $2 \log m$ random bits.

(b) This family is not universal. Consider the following inputs: $(x_1, x_2) = (0, 2^{k-1})$ and $(y_1, y_2) = (2^{k-1}, 0)$. We then have $h_{\alpha_1,\alpha_2}(x_1, x_2) = 2^{k-1}\alpha_2 \mod 2^k$ and $h_{\alpha_1,\alpha_2}(y_1, y_2) = 2^{k-1}\alpha_1 \mod 2^k$. Now notice that if α_2 is even (i.e. with probability $1/2$) then $h_{\alpha_1,\alpha_2}(x_1, x_2) = 0 \mod 2^k$ otherwise (if α_2 is odd) $h_{\alpha_1,\alpha_2}(x_1, x_2) = 2^{k-1} \mod 2^k$; likewise for α_1. So we get that $h_{\alpha_1,\alpha_2}(x_1, x_2) = h_{\alpha_1,\alpha_2}(y_1, y_2)$ with probability $1/2 > 1/2^k$, so the family is not universal.

(c) This family is universal. To see that, fix $x, y \in \{0, 1, \ldots, m-1\}$ with $x \neq y$. Now we need to figure out the following: how many (out of the $(m-1)^m$ in total) functions $f : [m] \to [m-1]$ will collide on x and y, i.e. $f(x) = f(y) = k$, for some fixed $k \in [m-1]$. Well, there are $(m-1)^{m-2}$ different functions $f : [m] \to [m-1]$ that have the property $f(x) = f(y) = k$ (because I just fixed the output of 2 inputs to some fixed $k \in [m-1]$ and allow the output of f for all other inputs to range over all $m-1$ possible values). Finally, ranging over all $m-1$ values of k, we get that there are $(m-1)^{m-1}$ functions $f : [m] \to [m-1]$ with the property $f(x) = f(y)$. So the probability of picking one such f is exactly $\frac{(m-1)^{m-1}}{(m-1)^m} = \frac{1}{m-1}$.

How many bits do we need in this case? Well, there is no succinct representation in this case, so we need to write down the whole family of functions explicitly and then pick one of the $(m-1)^m$ functions of the family. To do that we can imagine indexing all functions with integers $1, \ldots, (m-1)^m$ and randomly picking one such integer $k \in \{1, \ldots, (m-1^m)\}$; this obviously requires $\log(m-1)^m = m \log(m-1)$ bits.