Note 3: Streaming Algorithms

Acknowledgement: This note is based in part on an earlier version of Luca Trevisan.

Streaming algorithms are procedures whose input is presented in a continuous stream that must be processed sequentially in a single pass, using only very limited memory (much less than the size of the input). This setting corresponds to many real-life scenarios where the data to be processed is not stored anywhere but is generated dynamically, as is the case for requests to a server, online transactions, sensor data, and so on. Finite automata are the simplest examples of streaming algorithms, where the memory size is constant (equal to the number of states) regardless of the length of the input stream, and each input item is processed in constant time. In practice, however, we may allow our streaming algorithms a memory size that grows slowly (say, logarithmically) with the length of the input. This will already take us well beyond the capabilities of finite automata while remaining within the streaming framework. Our focus will be on memory size rather than on the time required to process items.

In this note, we briefly introduce the topic of streaming algorithms by seeing simple examples of problems that they can handle, along with lower bounds on the memory required for certain problems. The lower bounds will follow from an adaptation of the ideas we have already seen for finite automata.

We should mention that streaming algorithms are a very active research topic, especially in light of their connection to Big Data. However, since this is not the main concern of this course, we will only scratch the surface here.

1 A simple non-regular language

Let's start by returning to one of our canonical non-regular languages, namely

$$L_= = \{\text{set of 0-1 strings with an equal number of 0s and 1s}\}.$$

To put this in the streaming context, we assume that the binary input symbols $x_1, x_2, \ldots, x_n$ are presented in a stream, and that the algorithm must determine in a single pass whether the word $x_1 \ldots x_n$ belongs to L. We first make the simple observation that this problem can be solved using only $O(\log n)$ bits of memory: simply maintain a counter of the difference between the number of 0's and the number of 1's seen so far, and accept iff the counter is zero at the end of the stream. The range of this counter is clearly $[-n, n]$, so the number of bits required is $\lfloor \log_2 n \rfloor + 2$ (allowing one bit for the sign).

To see that this bound is optimal, we adapt the technology from our proof of the Myhill-Nerode theorem to the streaming setting.

Definition 1. *For a language $L \subseteq \Sigma^*$, we say that two streams $x, y \in \Sigma^*$ are* streaming distinguishable *for L* on inputs of length *n if there exists a stream $z \in \Sigma^*$ such that $|xz| = |yz| = n$ and exactly one of xz, yz belongs to L.*

This is essentially equivalent to our earlier definition of distinguishable strings for finite automata, except that we now require the lengths of xz, yz to be exactly n; this is because in the streaming scenario we care about the memory used as a function of the length of the input.

Theorem 2. *Suppose that there is a set $D(n)$ of pairwise distinguishable strings for L on inputs of length n. Then any streaming algorithm that recognizes L must use at least $\log_2 D(n)$ bits of memory on inputs of length n.*

Proof. Suppose, for a contradiction, that there is a streaming algorithm for L that uses $m(n) < \log_2 D(n)$ bits of memory on inputs of length n. Then this algorithm has at most $2^{m(n)} < D(n)$ distinct internal states, so by the pigeonhole principle there must be two different distinguishable strings x, y in our set that take the algorithm into the same state. But this in turn means that the output of the algorithm on inputs xz, yz must be the same, which is a contradiction. $\square$

Returning now to our language $L_=$, we need to find a large set of distinguishable strings S_n for strings of length n in $L_=$. From the definition, we note that any set of distinguishable strings must all have the same length. (Why?) Also, we may assume w.l.o.g. that n is even since there are no odd-length strings in $L_=$.

Consider the set $S_n = \{0^i 1^{n/2-i} : 0 \leq i \leq n/2\}$. We claim that these strings are all distinguishable: to distinguish $0^i 1^{n/2-i}$ and $0^j 1^{n/2-j}$, where $i \neq j$, we may use the string $z = 0^{n/2-i} 1^i$, since then $0^i 1^{n/2-i} z \in L$ and $0^j 1^{n/2-j} z \notin L$. Now $|S_n| = n/2 + 1$, so Theorem 2 implies that any streaming algorithm for $L_=$ must use at least $\lceil \log_2(n/2 + 1) \rceil$ bits of memory on inputs of any (even) length n. This implies that the above algorithm is optimal (up to one bit).

2 Finding a majority element

We turn now to a more challenging problem. Suppose our input is a stream of elements $x_1, x_2, \ldots x_n$ over some general alphabet Σ, and our goal is to find a *majority* element in the stream (i.e., an element that occurs more than $\frac{n}{2}$ times) Again, we want to do this in a single pass using only $O(\log n)$ bits of memory[1].

The imposed memory limitation basically allows us one (or a constant number of) counters up to about n; note that this rules out a trivial solution that keeps track of the frequencies of all elements, since this would require $|\Sigma|$ counters each potentially of size $O(\log n)$. We typically assume that Σ is very large (possibly even growing with n), so such a dependence on $\|\Sigma|$ is not reasonable. (Imagine, e.g., that Σ is the set of all Google search terms seen in a certain period.) How can we use a single counter to accomplish this task?

Exercise: Think about how you might solve this problem before reading on!!!

The idea is to maintain a counter for just one element as we move along the stream. Call this the *active* element. Initially there is no active element and the counter is set to zero. When the next element in the stream to be processed is x_i, we update the counter and active element according to the following rule:

- if the counter is zero then make x_i active and set the counter to 1, else
- [counter $\neq 0$] if x_i is active then increment the counter by 1, else
- [counter $\neq 0$ and some other element is active] decrement the counter by 1

At the end of the stream, we output the currently active element (if there is one). Note that this algorithm just needs to store the identity of one element and a single counter of maximum value n, so its memory requirement is only $\lceil \log_2 |\Sigma| \rceil + \lceil \log_2 n \rceil$ bits.

[1]Incidentally, this is a notoriously popular question in tech company interviews!

The correctness of the algorithm hinges on the following claim.

Claim 3. *If a majority element x exists in the stream, then x is active at the end of the process.*

Proof. Suppose x occurs $m > \frac{n}{2}$ times. If x is not active at the end, then we will show how to associate each occurrence of x with a *distinct* occurrence of some other element. This will be a contradiction, since there are fewer than m such other occurrences.

To show how to perform the above association, imagine that the counter is implemented using an explicit stack of occurrences of the currently active element: each increment pushes the new occurrence onto the stack, and each decrement pops the top occurrence off the stack. Now consider some particular occurrence of x in the stream: this occurrence causes the counter to be either incremented or decremented. If it causes a decrement, then we associate this occurrence of x with the occurrence of the currently active element (which must not be x) that is popped off the stack. On the other hand, if it causes an increment then this occurrence itself is pushed onto the stack, and we associate it with the later occurrence of some other element that causes it to be popped off the stack. (It must eventually be popped because of our assumption that x is not active at the end of the process.) This means that each of the m occurrences of x is associated with a distinct occurrence of some other element, giving us the desired contradiction. $\square$

Exercise: What, if anything, can you say about the output of the above algorithm in the case that the stream does *not* contain a majority element?

3 Finding a most frequent element

What if we don't assume that a majority element exists in our stream; can we still find an element that occurs most frequently using $O(\log n)$ space? Finding such an element is known as the "Most Frequent Element" (MFE) problem. As the following theorem shows, the answer is a devastating "no". We let $\ell = \lceil \log_2 |\Sigma| \rceil$ denote the bit length of the alphabet symbols, and as before n is the length of the input stream. For simplicity we will assume that $2^\ell > n^2$, which means that the size of the alphabet grows with n (at a rate of at least n^2).

Theorem 4. *Assuming $2^\ell > n^2$, any (deterministic) streaming algorithm for the MFE problem must use at least $\Omega(n\ell)$ bits of memory.*

Proof. We actually prove the lower bound for a simpler language-recognition problem which fits into our standard framework. Since this is a lower bound it will hold also for the original MFE problem as well. Denote the alphabet $\Sigma = \{0, 1, \ldots, 2^\ell - 1\}$. We consider the language L_{MFE} consisting of all strings over Σ for which the most frequent element is 0. We will show that there are $2^{\Omega(n\ell)}$ distinguishable strings for L_{MFE} on inputs of length n. This will prove the theorem via the general result of Theorem 2. (Note that here the alphabet Σ depends on n, but this does not affect Theorem 2.)

The set of distinguishable strings will be all strings of length $n-2$ that have the form

$$x_A = 0, 0, 0, a_1, a_1, a_2, a_2, \ldots, a_{(n-5)/2}, a_{(n-5)/2},$$

where $A = \{a_1, a_2, \ldots, a_{(n-5)/2}\} \subseteq \Sigma \setminus \{0\}$ ranges over all possible subsets of $\frac{n-5}{2}$ nonzero elements of Σ. Note that in x_A each element of A is repeated twice, while 0 is repeated three times and no other elements are present.

To see that any two strings x_A, x_B, where $A \neq B$, are distinguishable, let $c \in \Sigma$ be an element that belongs to A and not to B. (Such an element must exist since A, B are different subsets of the same size.) Then if we use $z = cc$ as the distinguishing string, we see that attaching z to x_A yields a string of length n that is not in L (because the most frequent element is c, which occurs four times), while attaching z to x_B yields a string of length n that is in L (because the most frequent element is 0, which occurs three times).

The number of such distinguishable strings is equal to the number of subsets A as above, which is

$$\binom{2^\ell - 1}{\frac{n-5}{2}} \geq \left(\frac{2^\ell - 1}{e \cdot (\frac{n-5}{2})}\right)^{\frac{n-5}{2}} \geq 2^{\Omega(n\ell)},$$

where we have used the standard fact that $\binom{N}{k} \geq \left(\frac{n}{ek}\right)^k$ and the fact that $\frac{2^\ell}{n} \geq 2^{\ell/2}$ which follows from our assumption $2^\ell > n^2$. □

4 A more general framework

We can actually relax the framework above to encompass a wider variety of inputs, as follows. For any given decision problem, define

$$L_{n,\Sigma} = \{\text{yes-instances of size } n\text{, encoded over } \Sigma\}.$$

Note that $L_{n,\Sigma}$ is just a language over the alphabet $\Sigma = \Sigma(n)$, corresponding to problem instances of "size" n. The actual length of the input stream may not be exactly n, but is related in a natural way to n.

As a concrete example, consider the problem of deciding whether a given undirected graph $G = (V, E)$ is connected. The set of inputs of size n here are all n-vertex graphs on the vertex set $V_n = \{1, 2, \ldots, n\}$. We can encode a given graph over the alphabet $\Sigma(n) = \{(v_1, v_2) : v_1, v_2 \in V_n, v_1 \neq v_2\}$ simply as the stream consisting of its edges (in arbitrary order). The language $L_{n,\Sigma}$ is defined as

$$L_{n,\Sigma} = \{G = (V_n, E)\text{: } G \text{ is connected}\},$$

where G is encoded as stated. Note that the length of the input stream for inputs of size n is $O(n^2)$.

As before, our goal is to design a streaming algorithm that recognizes $L_{n,\Sigma}$ for each n, and uses only a small amount of memory (say, $O(\log n)$ or $O(\text{polylog}(n))$). Conversely, to establish lower bounds on the memory size we can try to find a set $D(n)$ of pairwise distinguishable strings w.r.t. $L_{n,\Sigma}$. Note that, since problems of size n no longer have fixed stream length n, we say that two strings $x, y \in \Sigma(n)^*$ are *distinguishable* if there exists $z \in \Sigma(n)^*$ such that exactly one of xz, yz belongs to $L_{n,\Sigma(n)}$. Then, by exactly the same reasoning as in Theorem 2, we get a lower bound of $\log_2 D(n)$ on the number of bits of memory for any streaming algorithm for this problem.

We'll see a concrete example on HW4.