CS 341: ALGORITHMS (S18) — LECTURE 24

BONUS LECTURE II: THREE FUN PROBLEMS

ERIC BLAIS

In the CS 341 class, we covered a number of algorithm design and analysis techniques that are very useful for solving many different problems. But, perhaps just as important as the techniques themselves is the *idea* that the best way to solve many practical problems is to first formulate them as simple and clean abstract problems that we can then tackle effectively. In this last lecture, we cover three very different problems that are outside the scope of CS 341—and that are very different from all the problems we have seen so far—to see how we can do this.

1. Finding the median

We saw in a tutorial that there is a linear-time algorithm for finding the median in a set of n integers using the divide and conquer technique. This is great if all your data is stored locally—but what if the data is so large that it needs to be stored on 2 different servers? We can certainly run the same algorithm—perhaps on a third server that accesses the data on the 2 servers with queries made through the internal communication network. But then we will quickly find out that the resulting algorithm is extremely slow, and that the bottleneck for the algorithm is the amount of data that needs to be sent across the communication network: this algorithm requires a *linear* amount of communication, i.e., we essentially need to send the entire contents of at least one of the servers through the network, and this will take an enormous amount of time!

Can we do better? We have already done the first step in designing a more efficient algorithm: we have identified the bottleneck and most significant factor in the running time of any algorithm that solves the problem. In this case, it is the amount of communication required between the machines. (By comparison to the amount of time it takes to send any data across the network, local computation on any machine is essentially free.) Our next step is to see how we can define the problem we are trying to solve as precisely (and as simply!) as we can. We can do so in the *communication complexity* model of computation.

Definition 24.1. In the communication complexity version of the Median problem, two players named Alice and Bob hold sets of integers. Specifically, Alice has a list L_a of n integers in the range $\{1, 2, \ldots, n\}$ and Bob has a list L_b of n integers in the same range. They want to identify a median m of $L_a \cup L_b$, and the goal is to do so by designing a *protocol* (=algorithms run by Alice and Bob that can send messages to each other) that requires as few bits of communication as possible.

Note that here we *don't* worry about the time complexity of the algorithms, only the amount of communication exchanged between Alice and Bob. As we argued above, this is a realistic model of the cost of running algorithms on separate machines since communication will be orders of magnitude slower than any local computation.

There is a simple protocol that requires n bits of communication: Alice sends all her data to Bob, who then runs any median-finding algorithm on the joint data. Can you do better?

Theorem 24.2. *There is an algorithm that solves the communication complexity version of the* MEDIAN *problem and requires only $O(\log^2 n)$ bits of communication between Alice and Bob.*

Proof. The idea is to apply divide & conquer (or binary search) techniques. We can design a protocol where in each round, Alice and Bob know that the median lies in the range $[i, j]$ and, after the round, they know that the median lies in either $[i, \frac{i+j}{2}]$ or in $[\frac{i+j}{2}, j]$. Initially, Alice and Bob know that the median must lie in the range $[1, n]$ so after at most $O(\log n)$ rounds, they will have identified the median.

So let us now solve the single-round problem. Alice and Bob start the round both knowing the values of i and j. They can both compute $k = \lfloor \frac{i+j}{2} \rfloor$ (without using any communication!). Then Alice can count how many of her values are smaller than k and send that number to Bob, using $O(\log n)$ bits of communication. Bob can add that number to the number of his elements that are smaller than k: if the total is exactly n, then k is the median! If the total is less than n, then the median must be in the range $[k, j]$. And if the total is more than n, then the median must be in the range $[i, k]$. □

The protocol described above will run much, much faster than any MEDIAN algorithm that uses a linear amount of communication between the machines. You might be curious, however, if it is the *best* we can do for the problem. It's not: there is another protocol that only requires $O(\log n)$ bits of communication.

Challenge 1. *Design a communication protocol for the* MEDIAN *problem that requires only $O(\log n)$ bits of communication between Alice and Bob.*

2. HEAVY HITTERS

Here's a completely different problem: you're now in charge of all internet servers. You have a server (or a router) that handles requests, but because of possible attacks on your servers you want to notice whenever there is some source that is making too many requests. Here, we can define "too many" as, say, $n/100$ of all n queries in a day. The challenge here is that the requests are coming in fast, and you certainly don't have time or space to store much of the sources' information locally. And unlike all the other problems we have seen so far, we have no control over the order in which we access the data: we must process the requests in the order that they are given to us. This problem is known as the *heavy hitters* problem, and it can be formalized in the *streaming algorithms* framework.

Definition 24.3. In the HEAVYHITTERS problem, we are given some value k and we observe a sequence of positive integers $x_1, x_2, \ldots, x_n \in \{1, 2, \ldots, n\}$ in that order. We want to output a set S of at most k integers such that any integer that appears more than n/k times in the sequence will be in S. The goal is to do this while using as little memory as possible.

This problem is certainly an idealized version of the original problem. First, we simplified the problem itself by considering a sequence of integers instead of more complex objects (like source IP addresses). But if we solve this problem we'll see that it's very easy to replace "integers" with any other types of values in the future. Second, we don't say anything about time complexity—though in this case if we have a simple router we *do* want to make

sure we have an efficient algorithm. In this case, the motivation for the simplification is the observation that reducing the memory requirements of the algorithm appears to be the most challenging aspect of the problem, so it's best just to focus on that first. If we succeed in designing an algorithm that requires little memory, we can then focus on fast computation time afterwards. And third, you might notice that our requirement for a solution to the problem is weaker than what we might want in practice. (For instance, it might be important not to have any *false positives*; we might want *all* the entries in S to be present at least n/k times in the sequence.) This is again a simplification meant to help us obtain at least some progress on the original question: once we have solved this version of the problem, we can certainly revisit our solution to try to make it even stronger!

Let's again begin addressing the problem with a trivial solution: we can certainly solve the problem with n cells (so $O(n \log n) bits$) of memory: we store all the integers in the sequence locally, and then we run any algorithm that we like to find any heavy hitters in the sequence. But of course our goal is to do much better.

Theorem 24.4. *There is an algorithm that solves the* HeavyHitters *problem and requires only $O(k \log n)$ bits of memory.*

Proof. There can be at most k different heavy hitters in any sequence. The idea of the algorithm is to keep track of (at most) k candidate heavy hitters at all times along with a counter for each of these candidates. Then we run a simple greedy algorithm: whenever we see the next element x_i of the sequence we first check if it is one of the candidate heavy hitters. If so, we increment its counter. If it's not a candidate but we have space to store one more candidate, we add it to our list. Otherwise, we decrement the counter of each candidate currently in memory and expel any candidate that reached the count 0. The resulting algorithm is known as the *Misra–Gries algorithm*.

Algorithm 1: MisraGries(k, n)

```
A ← ∅;
for i = 1, 2, ..., n do
    if x_i ∈ A then
        count[x_i] ← count[x_i] + 1;
    else if |A| < k then
        A ← A ∪ {x_i};
        count[x_i] ← 1;
    else
        for each j ∈ A do
            count[j] ← count[j] − 1;
            if count[j] = 0 then
                A ← A \ {j};
return A;
```

The key observation in the proof of correctness of this algorithm is that if j is a heavy hitter (that occurs more than n/k times in the stream) then every time we decrement the count for j in the algorithm, we are also decrementing the count for $k-1$ other variables as well. And since every element in the sequence causes at most 1 counter increment and we have at least as many increment as decrement operations, it means that at most n/k

of the elements in the sequence can cause a decrement operation. Therefore, every heavy hitter element will be in A when the algorithm terminates. □

Challenge 2. *Improve the algorithm so that the set A returned by the algorithm satisfies two conditions:*

(1) *Every heavy hitter that appears more than n/k times in the sequence is present in A, and*
(2) *Every value returned in A is a "moderately heavy hitter" that appears at least $n/2k$ times in the sequence.*

3. Maximal independent set

One last problem in yet another completely different setting. We now have a very large set of sensors distributed over some environment. Two sensors *overlap* if their measurements are not independent of each other (e.g., if they are close to each other). We want to find a maximal set of sensors that do not overlap with each other. We don't have a centralized server connecting all the sensors, so the algorithm we design for this task has to be a *distributed algorithm*.

Note that for this problem the difference between *maximal* and *maximum* is quite important: we want a set S of non-overlapping sensors such that every other sensor overlaps with one of the sensors in S (so that we can't add to our current set without causing some overlaps), but S does not have to be the largest possible set of non-overlapping sensors.

The problem can be phrased as a graph problem.

Definition 24.5. In the MaxIndepSet problem, we have a graph $G = (V, E)$ on $|V| = n$ vertices that has maximum degree d. We run a (distributed) algorithm on each node such that each node can perform local computation and communicate with its neighbours. At the end of the computation, each node determines whether it is included in the final set $I \subseteq V$ or not. The final set I must be a maximal independent set in G, and our goal is to satisfy this condition while minimizing the (parallel) time complexity of the algorithm.

There is a very simple greedy algorithm for finding a maximal independent set in a graph: choose the vertex with the smallest label, add it to I, remove all its neighbours from the graph, and repeat. This algorithm is very efficient in the standard model of computation, but it does not parallelize: how could a node figure out that it is the one with the smallest label left in the graph without communicating with many other nodes in the graph?

Now, a small detour: we might get stuck on this question for a while and eventually throw our hands up in the air in frustration and decide that in the end, it's just much easier to go back to the centralized model and have a computer be in charge of knowing where all the sensors are and which ones overlap, so that it can compute the maximum non-overlapping set (or, abstractly: compute a maximal independent set of the overlapping graph) directly. Good idea! But given that our sensor networks will be quite large, we really would like our algorithm to *parallelize* effeciently, so that if we use an expensive multi-core computer as our centralized server, we do use all the cores effectively in our computation... but if that's what we want, we realize that we're right back to where we started: the simple maximal independent set algorithm will *not* parallelize efficiently, and the tasks of designing a distributed or a parallelizable algorithm for MaxIndepSet are in fact closely related to each other.

Theorem 24.6. *There is a randomized algorithm that solves the* MaxIndepSet *in expected time* $O(d \log n)$.

Proof idea. The idea of the algorithm is to divide up the computation into a series of rounds. In each round, all the nodes that are still active generate a random number between 0 and 1. The nodes share their generated number with all of their neighbours. If a vertex v has the minimal number among all its neighbours, it adds itself to I and all its neighbours decide that they won't be in I; all these nodes then become inactive. And we repeat this process until all the nodes become inactive. That's it! The standard pseudo-code for the algorithm would look like this:

Algorithm 2: Luby$(G = (V, E))$

```
I ← ∅;
while V ≠ ∅ do
    for each v ∈ V do
        r[v] ← random number in [0, 1];
        if r[v] < r[w] for each w ∈ N(v) then
            I ← I ∪ {v};
            V ← V \ ({v} ∪ N(v));
return I;
```

As a challenge: can you see how to write pseudo-code for the distributed version of the algorithm (that you would run on each node)? And how would you carefully describe the parallel version of the algorithm?

The correctness of the algorithm follows from the fact that you can never have two vertices connected by an edge add themselves to I—at most one of the two can have the minimum number $r[v]$ among all its neighbours, and when one does, the other node becomes inactive. The more challenging aspect of the analysis of this algorithm is the time complexity analysis. This is left as another exercise. □

Challenge 3. *Prove that each round of Luby's algorithm eliminates a constant fraction of the edges in the (active) graph in expectation. (Extra challenge: prove that this is true with high probability as well.)*