6.896 Sublinear Time Algorithms	September 9, 2004

Lecture 1

Lecturer: Ronitt Rubinfeld — *Scribe: Victor Chen*

1 Introduction

In this course, we study algorithms that run in time sublinear in the input. Since these algorithms can only process a very small portion of the input, one might think that such algorithms must necessarily be randomized and output some form of an approximation. In fact, suppose there is a problem for which there is a deterministic algorithm that runs in sublinear time and gives an exact answer. Then, the output cannot depend on any part of the input which has not been viewed. Surprisingly though, there are nontrivial problems for which a deterministic exact solution exists. In the following, we provide two examples of sublinear time algorithms that are contrary to the above characterization. The first is a problem that has a deterministic sublinear time algorithm which returns an exact solution. The analysis of the algorithm will be assigned for homework. The second is a problem that has a deterministic sublinear time algorithm which returns a good approximation.

1.1 Tournament

A *tournament* is a digraph such that for each pair of vertices u and v, exactly one of (u, v) and (v, u) is an edge. We can interpret the vertices as players such that each pair of players play a match, and an edge from one to another indicates that one player beats another, hence the name tournament. Assume that we have a tournament G on n vertices represented in adjacency matrix form. The size of the input is $\binom{n}{2}$. There exists a deterministic algorithm that finds the sink of G (a player who beats all others) if it exists in $O(n)$ time, which we will show in problem set 1.

1.2 Diameter

We have n points in a metric space. (Most importantly we need the axioms of symmetry and triangle inequality.) The input is a n by n distance matrix D such that $D(i, j)$ is the distance between i and j. We seek a sublinear time algorithm that outputs $\max_{i,j} D(i, j)$, the diameter. Here is a simple algorithm:

1. Pick a point u arbitrarily and output $z := \max_v D(u, v)$.

Clearly this algorithm runs in $O(n)$ time. Moreover, we argue that z, the value returned by this naive looking algorithm, is a good approximation for the diameter d of the input. More specifically, we claim that $d/2 \leq z \leq d$. To see this, let a and b be two points such that $D(a,b) = d$. Since D is a metric space,

$$d = D(a,b) \leq D(a,u) + D(u,b) \leq D(u,v) + D(u,v) = 2z.$$

We mention these two rare examples to see that not all sublinear time algorithms are randomized and provide approximate values. However, most sublinear time algorithms are both randomized and approximate, and this will be the focus of the rest of the course.

Note: Since we will analyze mostly randomized algorithms, we will make extensive use of various tail bounds, such as Markov, Chebyshev, Chernoff, and Hoeffding Bounds. Those unfamiliar with some basic probability theory should read the handout or look at the pointers on the class website.

2 Notions of Approximation

To study approximation algorithms, we need to define notions of how good an approximation is.

2.1 Optimization Problems

The following definition was implicitly used when we introduced the diameter problem, and is a commonly used characterization for optimization problems. We will use this and other variants throughout the course.

Definition 1 *Let $\pi(x)$ be the optimal solution of an input x. For $\gamma > 1$, we say that A is a γ-multiplicative approximation algorithm if for all x,*

$$\frac{\pi(x)}{\gamma} \leq A(x) \leq \gamma\pi(x).$$

We say that A is an α-additive approximation algorithm if for all x,

$$\pi(x) - \alpha \leq A(x) \leq \pi(x) + \alpha.$$

2.2 Decision Problems

To solve a decision problem, the algorithm accepts if the input has a certain property and rejects otherwise. A *property testing* algorithm accepts if the input has a certain property and rejects with high probability if the the input is "far" from the property, i.e., the input can be modified slightly so that it then has the property. This concept is best illustrated via an example.

Suppose we have a sequence of n numbers, $X_1, \ldots, X_n$. We would like determine if the sequence is monotonically increasing. A decision based algorithm accepts if X is monotonically increasing and rejects if it is not. Any decision algorithm runs in $\Omega(n)$ time to read the input and makes a decision. On the other hand, a property testing algorithm accepts if the sequence is monotonically increasing and rejects with probability greater than $2/3$ if more than ϵn of the X_i need to be removed so that the resulting sequence becomes monotonically increasing. Note that in this model, there is no restriction on what the algorithm should do if a sequence that is not monotonically increasing can become one with less than ϵn removals. In the remainder of the lecture, we exhibit a property testing algorithm for the sequence $\{X_i\}$ that runs in $O(\log n)$ time by selectively querying X_i.

Wlog, assume all X_i are distinct since we can interpret X_i as (X_i, i), which breaks ties without changing order. First, we consider some simple approaches that do not work. This will give us some intuition to designing a good sublinear time algorithm for this problem. The first approach is to select i randomly and test whether $X_i < X_{i+1}$. Consider the following sequence, which is very far from monotone:

$$1, 2, 3, \ldots, m, 1, 2, 3, \ldots, m, 1, 2, 3, \ldots, m, \ldots$$

Since this simple algorithm does not detect that the sequence is not monotonically increasing as long as it does not query a pair of locations of value m and 1 respectively, this approach will need $\Omega(m)$ queries. Now consider a more sophisticated algorithm which samples the sequence at random points and checks if the random points form a monotonically increasing sequence. However, consider the following sequence, which is again very far from monotone:

$$m, m-1, \ldots, 1, 2m, \ldots, m+1, 3m, \ldots, 2m+1, \ldots$$

The algorithm detects that this sequence is not monotonically increasing only if two of its query points fall within $[km, (k-1)m+1]$ for some k. By the Birthday Paradox, this is unlikely if the number of samples is $o(\sqrt{n})$ and m is a constant (m should be bigger than 2, so that the distance of the sequence from monotone is at least $1/2$). With high probability, the values of the query points will form a monotonically increasing subsequence. In a sense, since the distance between each random sample is roughly the same, the random points are ignorant of the changes between each pair of points.

Here is another approach that avoids the previous pitfall by using binary search.

SortedTest$(X_1, \ldots, X_n, \epsilon)$
Repeat Steps 1 to 3 $O(1/\epsilon)$ times:
1. Pick i uniformly random from 1 through n.
2. Query X_i.
3. Perform binary search for X_i. If the search does not end up in the ith

or that the sequence is revealed not to be monotonically increasing during the search, reject.

4. Accept if all searches are successful.

Correctness of the algorithm:

The algorithm runs in time $O(\log n/\epsilon)$ since each binary search takes $O(\log n)$ time. If the sequence $\{X_i\}$ is monotonically increasing, then clearly the algorithm accepts. We need to show that if more than ϵn of the sequence need to be removed for it to be monotonically increasing, then the algorithm rejects with probability more than $2/3$. Suppose not, that the algorithm accepts with probability at least $1/3$. We call index i good if a binary search for X_i finds X_i in location i with no inconsistencies during the search. Then at most ϵn of the indices are bad; otherwise, each time through the loop, the algorithm finds a bad index with probability at least ϵ. Then the algorithm accepts with probability at most $(1-\epsilon)^{c/\epsilon} = e^{-c} < 1/3$ for some constant c, a contradiction. Hence, at most ϵn indices are bad, and we can simply delete them. Now all the remaining points are good, and it remains to see that the good points form a monotonically increasing sequence. Say i and j are good and $i < j$. Consider the first point in the binary search path where X_i and X_j diverge and assume that point has value u. Since i and j are good and $i < j$, we can conclude $X_i \leq u \leq X_j$. This concludes the proof.