6.046 Recitation 12

May 11, 2012

Plan for recitation:

- Problem(s) for sublinear algorithms
- Problem for clustering

1 Sublinear Algorithms

We expect the lower bound for any algorithm to be linear time – in general, in order to do reasonable things with the input, it has to read all of the input. But we can get more efficient algorithms by sacrificing correctness.

Motivation:

- the data we're dealing could be so large that linear time algorithms are still too slow
- the boost in efficiency might just be enough to make the lowered correctness worth it
- can quickly filter down to a smaller set of things you have to check

In lecture, we did:

- Number of connected components in a graph
- Size of a minimum spanning tree

1.1 Testing Monotonicity

Given a list $x_1, x_2, \ldots x_n$ of items, are they in sorted order?

In order to solve this deterministically, we have to take $\Omega(n)$ time – there's no way to avoid looking at every single element (otherwise, the one we miss might be the out-of-order one).

Goal: Find a sublinear algorithm to test for monotonicity.

Definition (ϵ-far). An list of length n is ϵ-far from monotonic if we must delete $\geq \epsilon n$ elements from the list for the remaining elements to be in sorted order.

We will design an algorithm that is always correct if the list is monotonic, and incorrect with probability at most 1/3 if it is ϵ-far from monotonic. (Behavior of the algorithm is undefined if it is not monotonic but not ϵ-far; the idea is that we're okay with being wrong if it's that close anyway.)

1.1.1 Attempt 1: Naive algorithm.

We take a uniform sample of s indices and check whether that set is monotonic. Note that with a samule size of s, checking whether those items are sorted will take $O(s)$ time.

This approach will take $\Omega(\sqrt{n})$ time. Why?

Consider the list such that $x_i = i + 1$ for odd i and $x_i = i - 1$ for even i. In other words, the list contains $2, 1, 4, 3, 6, 5, \ldots, n/2, n/2 - 1$.

Q. What is ϵ for this list? In other words, how far is it from monotone?

Answer: it is 1/2-far from monotone, because we have to remove every other item in the list (if even one pair of swapped elements stays, then it will be out of order).

If the subset of elements we select has a pair of swapped elements (contains both elements of index $2j$ and $2j + 1$), then the algorithm will return No.

Claim. If the sample size $s \leq \sqrt{n}/2$, then the probability of the sample containing an out-of-order pair is less than 1/3.

Proof. First, if we are given any pair of elements, then what is the probability that they are a pair?

There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs in total, and of those, $\frac{n}{2}$ are out-of-order pairs. So the probability of getting an out-of-order pair is $\frac{1}{n-1}$.

Within the sample set, which is of size s, there are $\binom{s}{2}$ pairs, which means that the probability of at least one pair being out-of-order in this set is at most $\binom{s}{2}\frac{1}{n-1}$.

We are given that $s \leq \sqrt{n}/2$, so we have $s(s-1) \leq (n-1)/4$. Then $\binom{s}{2}\frac{1}{n-1} \leq \frac{1}{8} < \frac{1}{3}$.

So the algorithm will be correct with probability less than 1/3 on this input; therefore in order to always have a probability of correctness that is greater than 2/3, we require a sample size that is larger than this one, which implies that it will take $\Omega(n)$ time.

If we proved an upper bound, this would be sublinear, but we can do better than $\sqrt{n}$.

1.1.2 Attempt 2: Better algorithm.

New approach: pick an element x_i, which is the element at index i. Perform a binary search for x_i. Define index i to be *bad* if any index along this binary search fails monotonicity (in other words, if $i < j$ but $x_i > x_j$), and *good* otherwise.

We'll repeat this c/ϵ times to get a $O(\frac{1}{\epsilon} \lg n)$ time algorithm.

Correctness: It's clear that if the input list is monotonic, there will be no bad indices, and therefore this algorithm will always accept.

It remains to be shown that, if it is ϵ-far from monotone, then the algorithm is correct (return NO) with probability 2/3.

Claim. If the input is ϵ-far from monotone, then there are $\geq \epsilon n$ bad indices.

Proof. We show the contrapositive by showing that if there are $< \epsilon n$ bad indices, then there is a monotonic subsequence of length $\geq \epsilon n$.

We can show that, for any i, j which are both "good" indices, it must be the case that $x_i < x_j$.

Let k be the index just before where the binary searches for x_i and x_j diverge. This is equivalent to finding the least common ancestor of x_i and x_j in the binary search. Thne we know that $i < k < j$, since both i and j are good indices, and correspondingly that $x_i < x_k < x_j$.

Finally, the probability that we choose all good indices when there are $\geq \epsilon n$ bad indices is $< (1 - \epsilon)^{c/\epsilon} < e^{-c}$, and we can pick c accordingly such that we have an appropriate bound for the probability of error.

1.2 Testing for Unimodal Lists

(From the fall 2010 final). A list A is unimodal if there exists some index k such that $A[1..k]$ is monotonically increasing and $A[k..n]$ is monotonically decreasing.

A is ϵ-far from unimodal if removing at most ϵn elements would result in the remaining elements being unimodal.

Goal: find a sublinear time algorithm to test whether A is unimodal.

Strategy:

1. Use binary search to find k. At each step of the binary search, test two adjacent elements to see if they are increasing or decreasing, which tells us whether we're to the left or to the right of k.

2. Use the monotonicity checker we developed previously on each of $A[1..k]$ and $A[k..n]$, and return YES if both of the sub-problems return YES. (We now assume that we can specify the direction of monotonicity.) Because the monotonicity checker always outputs YES when the input is monotonic, if A is unimodal then we are guaranteed to get the right answer. If A is ϵ-far from being unimodal, then one of its sides must therefore be ϵ-far from being monotonic. The algorithm run on that side will return NO with probability at least 2/3, and therefore the overall algorithm will return NO (which is the correct answer) with probability at least 2/3.

2 Clustering

Recall from class that we covered two kinds of clustering:

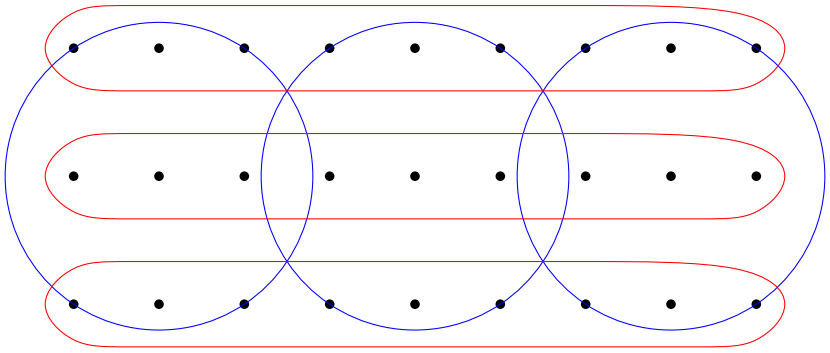
1. Hierarchical Agglomerative Clustering: given a set of points x_i and a distance metric $d(x_i, x_j)$ describing the distance between a pair of points, and a target number of clusters k, return a clustering of the points that essentially maximizes the distance between the different clusters.
2. Min-radius clustering: given a set of points x_i, a distance metric (with triangle inequality), a target number of clusters k, and a radius r, choose a set of k points such that every point is at most distance r from one of those points. (This is NP-hard!)

We defined an approximation of the min-radius clustering problem to be one that approximates the radius, while keeping the number of clusters fixed.

2.1 Short clustering question

Show that, in general, the k clusters found using hierarchical agglomerative clustering do not provide a 2-approximation to the minimum radius clustering problem.

Solution: Here is a counterexample, with $k = 3$, where the hierarchical agglomerative clusterig algorithm does not provide a 2-approximation to the min-radius clustering. The clusters that result from hierarchical agglomerative clustering are circled in red, while the results of the min-radius clustering are circled in blue.



6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012