Sublinear Time Algorithms

As stated in lecture, the idea behind these algorithms is to relax our notions of correctness and examine a small, random sample of the input in order to run quickly on massive data sets. Today we will look at another sublinear time algorithm for checking clusterability.

First let's formalize the definition of graph clusterability. We will define the set of points $P \subseteq \mathbb{R}^n$ as (K, R)-radius clusterable if there is a size K subset $C \subseteq P$ such that each point in P is within a distance of R of some element of C.

I will note but not prove that deciding if a set of points is (K, R)-radius clusterable ((K, R)-r.c.) is NP-Complete, and has a 2-approximation algorithm. I will also note that this formulation of clusterability cannot be checked in sublinear time, since a single point may make the entire set not (K, R)-r.c. Therefore we're going to change our goal slightly.

Definition: Let the set of vertices P be ϵ-close to (K, R)-r.c. if one can delete $\epsilon\,|P|$ points to make it (K, R)-r.c. Otherwise P is ϵ-far from (K, R)-r.c.

Like most sublinear time property testers, there are some inputs on which the tester will say "Yes," some inputs on which it will say "No," and some inputs in between on which we don't care what it says. Our clustering property tester will test the following: On input P, K, R, ϵ, if P is (K, R)-radius clusterable, it will output "Yes." Otherwise, if P is ϵ-far from $(K, 2R)$-r.c., it will output "No" with probability $2/3$. Note that if we repeat this algorithm $3/2 \ln 1/\beta$ and pass only if all trials pass, then we output "No" with probability better than $1 - \beta$.

Algorithm: Begin with an empty set of points C. Repeat the following at most $K + 1$ times: On each iteration, pick a sample S of size $\frac{\ln 3(K+1)}{\epsilon}$ points from P. Check each point in S against each point in C. For each point $x \in S$ such that $dist(x, C) > 2R$, add x to C. If the size of C ever exceeds K, output "No." Otherwise output "Yes."

Correctness:

We must verify the two property testing requirements for this algorithm. First we will verify that if the given set of points is (K, R)-r.c., this algorithm always outputs "Yes." Second, we will verify that if P is ϵ-far from $(K, 2R)$-r.c., this algorithm will output "No" with probability better than 2/3.

We will prove that if this algorithm outputs "No," then P cannot be (K, R)-r.c. Suppose the algorithm outputs "No." Then we have identified $K + 1$ points that all have pairwise distance at least $2R$. By definition of (K, R)-r.c., this cannot happen if P is (K, R)-r.c. By taking the contrapositive of this statement, if P is (K, R)-r.c., then this algorithm must output "Yes."

Now we will show that if P is ϵ-far from $(K, 2R)$-r.c., then this algorithm outputs "No" with probability better than 2/3. The idea behind this proof is that if P is ϵ-far from $(K, 2R)$-r.c., then there must be a lot of outliers. Furthermore, if there are a lot of outliers, we will prove that each iteration probably increases the size of C, and therefore C will be too big after $K + 1$ iterations.

We first show that if P is ϵ-far from $(K, 2R)$-r.c., then there must be a lot of outliers. Define $x \in P$ to be a candidate if the distance between x and any point in C is greater than $2R$. We claim that on each iteration, if $|C| \leq K$, then there must be more than ϵn candidates. Assume that this is not the case, and thus on some iteration there are at most ϵn candidates. If we remove all of these candidates, then all remaining points are within $2R$ of some point in C. Consequently, P is, by definition, ϵ-close to $(K, 2R)$-r.c., which contradicts our original assumption. Therefore, if P is ϵ-far from $(K, 2R)$-r.c., then it must have more than ϵn candidates.

Now we will show that this algorithm will probably see enough such candidates over all iterations if P is ϵ-far from $(K, 2R)$-r.c. This might not be the case if some candidate is not seen on any iteration. Therefore, we can bound the probability that not enough candidates are seen by the probability that some candidate is never sampled. Consider any single iteration of this algorithm. The probability that in some iteration no candidate is sampled is at most $(1 - (\epsilon n)/n)^{\frac{\ln 3(K+1)}{\epsilon}}$. Using the property that $(1 - 1/k)^k$ is approximately $1/e$ as k goes to ∞, This probability is at most $(1/e)^{\ln 3(K+1)} = \frac{1}{3(K+1)}$. Now, by bounding the probability that some candidate is never sampled by this algorithm using the union bound, we observe that this probability is at most $(K + 1)\frac{1}{3(K+1)} = 1/3$. Consequently, the probability that this algorithm does not see enough candidates over all iterations is at most 1/3, and the probability that it does find enough candidates to output "No" is at least 2/3. Therefore, this algorithm fulfills the property testing requirements.

Runtime:

On each iteration we compare ever point sampled to every point in C, which is $\frac{\ln 3(K+1)}{\epsilon} * (K + 1) = O(\frac{K \ln K}{\epsilon})$ comparisons. This algorithm executes at most $K + 1$ iterations, so the total runtime is $O(\frac{K^2 \ln K}{\epsilon})$. This runtime is independent of the number of points, and therefore we have a sublinear time algorithm.

Streaming Algorithms

Another interesting sublinear situation involves processing a stream of data. A stream is a long sequence of n values, where n is sufficiently large that our algorithm only gets a single pass over the data. In the stream model of computation, n is sufficiently large that we cannot store it entirely in memory, and in fact the size of our memory is usually $O(\log n)$. Furthermore, we must process each element quickly.

To present some notion of what may be computed in this model, its best to look at an example. Suppose we have a stream of integers in the set $\{1, 2, \cdots, m\}$ and we want to identify the element that occurs more than half of the time in the input sequence, which we call the majority element. (For this case we may assume that we know such an element exists.) How might we do this?

We can find the majority element using a single pass over the stream and maintaining a single value and a counter. When we encounter an element for the "first" time, we will store that elements value and set the counter to 1. In subsequent reads from the stream, if we encounter that same element, we will increase the counter. If, however, we encounter a different element, we will decrement this counter. Whenever the counter reaches 0, we will evict that element from memory and store the current element in the stream with a count of 1. Note that storing an element requires $O(\log m) = O(\log n)$ space, and storing a counter requires $O(\log n)$ space, so this algorithm fits the streaming model of computation.

We can prove this algorithm will find the majority element. On each read from the stream we will describe the modification to the counter as either a modification in favor of the majority element or a modification against the majority element. If the majority element is currently stored in memory, incrementing the counter favors the majority element, while decrementing the counter goes against the majority element. If the majority element is not currently stored in memory, decrementing the counter favors the majority element, and incrementing the counter goes against the majority element. The number of modifications in favor of the majority element is at least the number of occurrences of the majority element, while the number of modifications against the majority element is the number of occurrences of something else. By definition of majority element, this first number is larger than the second, so this algorithm must terminate with a counter in favor of the majority element. Therefore this algorithm is correct.