Sublinear Algorithms Notes (Lec 22)

Classic Algorithms
Read the input, compute an answer.

Online Algorithms
Eventually read the input, provide answers after each read piece.

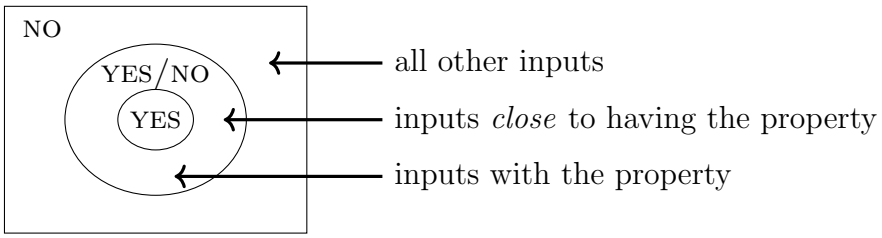
Streaming Algorithms
Eventually read the input, but only remembers a tiny amount of what has been read and use that to compute the answer.

Sublinear Algorithms
Compute an answer in much less time than is needed to read the input (i.e. read a tiny part of the input and compute the answer). Very useful if the data is huge.

Is this even possible? Say that you want to check if an input graph has a triangle. Don't you need to read the whole graph? As a bad input, consider (a) a complete bipartite graph and (b) a complete bipartite graph with one added edge. (a) has no triangle and (b) does not. Any algorithm (even randomized) must query a constant fraction of the vertex pairs to distinguish (a) and (b).

The solution? Settle for approximately correct answers! There are two interpretations of 'approximate':

- **Classic Approximation.**
  Return an answer that is within α (*approximation factor*) of the correct value. This is useful for optimization problems but not useful for decision problems.
- **Property Testing.**
  Return YES if the input is a YES-instance, or sometimes if it is *close* to a YES-instance.
  Return NO if the input is *far* from a YES-instance (or sometimes if it is close).



Example of Classic Approximation: Diameter of a Point Set

Input.
An n-by-n distance matrix $\mathbf{D}$ for which

- $D_{ij} = D_{ji}$ for all i, j (Symmetry).
- $D_{ij} \leq D_{ik} + D_{kj}$ for all i, j, k (Triangle Inequality).

Output.
An estimate of the *diameter* $d_* = \max_{i,j} D_{ij}$.

Algorithm.

(1) Pick an arbitrary point $p \in \{1, \cdots, n\}$.

(2) Compute $\hat{d}_* = \max_{1 \le i \le n} D_{pi}$ and output $\hat{d}_*$.

Runtime. $O(n) = O\left(\sqrt{N}\right)$ (square root of the input size!)

Correctness.
Say $d_* = D_{i^*j^*}$. Then $D_{i^*j^*} \le D_{i^*p} + D_{pj^*} \le 2\hat{d}_*$. Hence $0.5d_* \le \hat{d}_* \le d_*$ (2-approximant). □

Remark.
2-approx is *optimal* using a sublinear algorithm. Consider $\mathbf{D}$ being all-ones, and $\mathbf{D}'$ being all-ones except $D'_{ij} = D'_{ji} = 2$ for some i, j. Any algorithm must access either (i, j) or (j, i) to distinguish $\mathbf{D}'$ from $\mathbf{D}$. Since (i, j) is unknown, $\Omega(n^2)$ queries are necessary to be better than 2-approx!

Example of Property Testing: Graph Connectedness

Definition. A graph G on n vertices and maximum degree Δ is *ε-close to connected* if adding $< \varepsilon n\Delta$ edges can make G connected. *Remark:* We assume $\varepsilon > 1/(n\Delta)$ so $\varepsilon n\Delta > 1$.

Input.
An *adjacency list* of a graph G with n vertices and maximum degree Δ, and a parameter $0 < \varepsilon < 1$.

Output.
If G is connected, output YES. If G is not ε-close to connected, output NO with probability ≥ 0.99. (Note: If G is not connected but ε-close, we don't care what it outputs.)

Main idea. If G is not ε-close to connected, we must add $\ge \varepsilon n\Delta$ edges to connect it, so

- G has many connected components ($> \varepsilon n\Delta$).
- At least half of these components are *small*, i.e. $< \frac{2}{\varepsilon\Delta}$ nodes.
- Many vertices are in small components.

Therefore, we first run DFS on some random vertex until either (a) a small component is found (not connected), or (b) enough vertices are reached (large component), and then repeat.

Algorithm. We pick c later in the analysis.

(1) Repeat $\frac{c}{\varepsilon\Delta}$ times:

  (1) Pick a random node s and run DFS until either
    (a) $\ge \frac{2}{\varepsilon\Delta}$ distinct nodes are seen.
    (b) or s is in a component of size $< \frac{2}{\varepsilon\Delta}$. In this case, immediately return NO.

(2) Return YES.

Runtime.

$$\underbrace{\frac{c}{\varepsilon\Delta}}_{\#\text{ iterations}} \times \underbrace{\left(\frac{2}{\varepsilon\Delta} \times \Delta\right)}_{\text{BFS (\# edges)}} = O\left(\frac{1}{\varepsilon^2\Delta}\right).$$

Correctness. We first formalize the Main Ideas:

- **Claim 1.** If G is not ε-close to connected, then G has $> \varepsilon n\Delta$ components.
  Proof. If it had $\leq \varepsilon n\Delta$ components, connect them in a straight line with $\varepsilon n\Delta - 1$ edges! □
- **Claim 2.** If G has $> \varepsilon n\Delta$ components, it has $\geq \frac{\varepsilon n\Delta}{2}$ components of size $< \frac{2}{\varepsilon\Delta}$ (*small*).
  Proof. If not, there will be $> \varepsilon n\Delta - \frac{\varepsilon n\Delta}{2}$ components of size $\geq \frac{2}{\varepsilon\Delta}$, giving $> n$ vertices! □
- **Claim 3.** If G is not ε-close to connected, G has $\geq \frac{\varepsilon n\Delta}{2}$ nodes that are in small components.
  Proof. Use claim 2 and that each component has at least one node. □

We now analyze the algorithm.

- If G is connected, the algorithm returns YES because there are no vertices in small components.
- If G is not ε-close to connected, the claims imply

$$\Pr_{s\in V}[s \text{ is in a small CC}] \geq \underbrace{\frac{\varepsilon n\Delta}{2}}_{\#\text{ nodes in small CC}} \times \underbrace{\frac{1}{n}}_{\text{being sampled}} = \frac{\varepsilon\Delta}{2}.$$

$$\therefore \Pr[\text{alg returns NO}] \geq 1 - \left(1 - \frac{\varepsilon\Delta}{2}\right)^{c/\varepsilon\Delta} \geq 1 - e^{-c/2}$$

  which can be chosen to be ≥ 0.99.

Example of Property Testing: List Sortedness

Definition. A list of size n is *ε-close to sorted* if we can delete $\leq \varepsilon n$ items to get a sorted list.

Input.
A list $L = (x_1, \cdots, x_n)$ of *distinct* numbers, and a parameter $0 < \varepsilon < 1$.

Output
If L is sorted, output YES. If L is not ε-close to sorted, output NO with probability ≥ 0.99.

Wrong ideas.

- Picking a random i and check if $x_i < x_{i+1}$. Bad example: $(1, 3, 5, 7, \cdots, \frac{n}{2}, 2, 4, 6, 8, \cdots)$
- Picking random $i < j$ and check if $x_i < x_j$: Bad example: $(2, 1, 4, 3, 6, 5, 8, 7, \cdots)$

Algorithm. We pick c later in the analysis.

(1) Repeat $\frac{c}{\varepsilon\Delta}$ times:

  (1) Pick a random $1 \le i \le n$.

  (2) Binary search for x_i in the list L as if L were sorted.

  (3) If we find an inconsistency during the search or we don't end up at location i, return NO.

(2) Return YES.

Example. Consider $L = (1, 3, 5, 7, 2, 4, 6, 8)$.

- $i = 6$: We binary search for $x_6 = 4$.

  Compare with 7 $\to$ Compare with 3 $\to$ Get 5, return NO.

- $i = 3$: We binary search for $x_3 = 5$.

  Compare with 7 $\to$ Compare with 3 $\to$ Get 5, continue loop.

- $i = 8$: We binary search for $x_8 = 8$.

  Compare with 7 $\to$ Compare with 4 !! return NO.
  (Inconsistency! We are looking at the halflist with values > 7 but met 4)

Runtime.

$$\underbrace{\frac{c}{\varepsilon}}_{\#\text{ iterations}} \times \underbrace{O(\log n)}_{\text{BFS (\# edges)}} = O\left(\frac{\log n}{\varepsilon}\right) << n.$$

Correctness.
Define an index i to be *good* if the binary search for x_i succeeds, otherwise *bad*.

Key Claim. If $i < j$ are both good, then $x_i < x_j$.

Proof. Consider the sublist of elements that are encountered by the binary search for x_i or for x_j. This sublist has consistent binary search, so it is sorted and hence $x_i < x_j$. □

As a corollary, removing all bad indices gives a sorted list! Hence if L is not ε-close to sorted, the number of bad indices is $> \varepsilon n$. We now analyze the algorithm.

- If L is sorted, all indices are good so YES is always returned.
- If L is not ε-close to sorted, then there are $> \varepsilon n$ bad indices, so

$$\Pr[\text{sampling a bad index}] > \varepsilon n \times \frac{1}{n} = \varepsilon.$$
$$\Pr[\text{alg returns NO}] \ge 1 - (1-\varepsilon)^{c/\varepsilon} \ge 1 - e^{-c}$$

  which can be chosen to be ≥ 0.99.

Want to know more? Take **6.5240**.