COMS E6998-9: Algorithms for Massive Data (Spring'19) Mar 14, 2019

Lecture 16: Monotonicity and Sublinear Time Algorithm for Graphs

Instructor: *Alex Andoni* Scribes: *Wei Liu, Matthew Lerner-Brecher*

1 Introduction

In today's lecture, the first half focused on monotonicity testing algorithm, whereas sublinear time algorithm for graphs is presented in the second half.

2 Monotonicity Testing

2.1 Problem Statement

Input $x \in$ [1, 2, ..., n] are integers, the goal is to distinguish if x is sorted (monotonically increasing), or "ϵ-far" from sorted.

Being ϵ-far means that in order to make x sorted, one needs to delete $\geq \epsilon n$ positions. In other words, the longest increasing subsequence is $\leq (1-\epsilon)n$.

Theorem 1. *It requires $O(\frac{\log n}{\epsilon})$ queries*

There are other definitions of ϵ-far. For example, ϵ-far-pairs means that there are $\geq \epsilon n^2$ of pairs (x_i, x_j), where $i < j$, but $x_i > x_j$.

It can be detected if x satisfies $\geq$ ϵ-far-pairs property by checking the number of random pairs (x_i, x_j) is of order $\frac{1}{\epsilon}$

Suppose we have an algorithm that picks random $i < j$ and checks if $x_i < x_j$. This algorithm would not perform well in a sequence that simply switches the neighboring pairs in a sorted stream, i.e., [2, 1, 4, 3, 6, 5......]. To counter such inefficiency, one might propose to pick a random i and let $j = i + 1$. Such an algorithm would not efficiently detect a sequence such as [$\frac{n}{2}$, $\frac{n}{2}+1$, ..., 1, 2, 3, ...] to be unsorted.

Algorithm 1. *Iterate $\frac{3}{\epsilon}$ times. For each iteration, pick $i \in [n]$, perform binary search on $y = x_i$ between $[s, t]$ (starting from $[1, n]$), $m = \lfloor \frac{s+t}{2} \rfloor$. It fails if 1) binary search is inconsistent (i.e. $x_s < x_m < x_t$ is false); 2) y not found. Finally, output "sorted" iff there's no fail.*

Analysis 1.

1. run time: $O(\frac{1}{\epsilon} \log n)$ queries
2. if x is sorted, then the algorithm will output sorted with probability 1
3. if x is ϵ-far from sorted, then the algorithm will output not sorted and if we can show $Pr[an\,iteration\,fails] \geq \epsilon$, then $Pr[all\,iterations\,succeed] \leq (1-\epsilon)^{\frac{3}{\epsilon}} < e^{-\epsilon\frac{3}{\epsilon}} = e^{-3} < 0.1$

Claim 2. $Pr[an\,iteration\,fails] \geq \epsilon$

Proof. Definition: $i \in [n]$ is "good" if binary search on i succeeds.

Suppose i and j are both good. Let m denotes the least common ancestor of i, j in binary search tree.

Since i is good, we have $x_i < x_m$; since j is good, we have $x_m < x_j$. Therefore, we can conclude $x_i < x_j$, which means that all good i's are in sorted order. In other words, there must be $c(1-\epsilon)n$ good i's. □

2.2 Remarks

Remark 1. It is possible to remove the assumption of distinctiveness by transforming the original sequence x into x', where $x'_i = (x_i, i)$. Sorting in the new sequence x' becomes $x'_i < x'_j$ iff $x_i < x_j$ or $x_i = x_j$ & $i < j$

Remark 2. The algorithm is adaptive, meaning that each step depends on the outcome of the previous step. However, we can make the algorithm non-adaptive: we can generate the entire list of queries at the beginning before seeing any answers.

Non-adaptive version of the algorithm: our observation is, if x is sorted, for any fix i, we know the sequence of queries. Therefore, we generate all expected queries. If the algorithm fail to find the item, then the sequence is ϵ-far from sorted.

Algorithm 2. *(repeat $O(\frac{1}{\epsilon})$ iterations) Pick i, for each $j = 2^e$, where $e = 1, 2, \log n$. Pick $k \in [i-j, i+j]$, check if x_i and x_k in the right order.*

Remark 3. The $\log n$ factor is optimal.

Remark 4. A more general version of this problem that is commonly studied is to determine whether $f : [n]^d -> [0,1]$ is monotone (that is if $\vec{x}$ dominates $\vec{y}$ then $f(\vec{x}) \geq f(\vec{y})$).

3 Sublinear Graph Algorithms

Our goal here will be to deduce properties about a graph G with better than $O(n)$ time.

3.1 Types of Graph Representations

The kind of graph representation used will be important for being able to get sublinear time algorithms. The two usual graph representations are:

1. Adjacency Matrix - This representation is better for dense graphs (where # edges $\approx n^2$). This is because getting a list of the edges is already linear time, so we cannot easily pick a random edge unless it's dense.

2. Adjacency List - This representation is better for sparse graphs. This is because picking a random edge and hence neighbor no longer takes linear time. However, we no longer can easily determine as easily if there is an edge between two vertices.

3.2 Types of Graph Questions

Questions about graphs will often use the following definition:

Definition 3. *We say G is "ϵ-far" from connected if we need ϵn edges to make the graph connected.*

We can similarly define ϵ-far for properties other than connectedness such as triangle free (this latter property is actually connected to graph regularity for dense graphs). The questions we'll study will be of the two forms:

1. Testing Questions - These are essentially questions with a yes or no answer. For example "is G connected" versus "G is ϵ-far from connected"

2. Property Estimate - Essentially questions of the form compute up to approximation the number of CC (connected components), the MST value, and vertex cover.

3.3 MST Value Estimation

For this section assume G is a connected graph such that all vertices have degree $\leq d$. Our goal is to show the following theorem about the minimum spanning value:

Theorem 4. *If the weights of G are in $\{1, 2, \dots, M\}$ then we can compute the MST value up to a $1+\epsilon$ factor in $O\left(\frac{M^4 d}{\epsilon^3}\right)$ time.*

Main Approach 1. *We will reduce the MST-Value computation to determining the number of connected components.*

For example

- For $M = 1$, the answer is obviously always $n-1$ as every node in the spanning tree has weight 1

- For $M = 2$, we apply an algorithm inspired by Kruskal's. Let G_1 be the subgraph of G where we only include edges of weight 1 and let CC_1 be it's connected components. We need at least $CC_1 - 1$ edges of weight 2 to connect the components of CC_1. However, for the other $n - CC_1$ edges we can take vertices of length 1 by taking a MST on each connected component. Thus in total the MST value is:
$$(n - CC_1) + 2(CC_1 - 1) = n - 1 + CC_1 - 1$$

The approach here generalizes:

Fact 5. *Let G_i be the subgraph of G consisting of edges of weight $\leq i$ and let CC_i be it's number of connected components. Then*
$$MST = n - 1 + \sum_{i=1}^{M-1} (CC_i - 1)$$

Proof. By the same reasoning as used in the $M = 2$ case, the minimum number of weight k edges we need will be $CC_{k-1} - CC_k$ as we need that many edges to connect the C_{k-1} components to each other.

Using this we get a telescoping sum

$$\begin{aligned} MST &= (n - CC_1) + 2(CC_1 - CC_2) + \cdots + (M-1)(CC_{M-2} - CC_{M-1}) \\ &= n - 1 + \sum_{i=1}^{M-1} (CC_i - 1) \end{aligned}$$

□

Idea 1. *To estimate the MST value we will estimate each CC_i value up to a factor of $\delta = \frac{\epsilon}{M}$.*

Idea 2. *To estimate the number of connected components in a graph H we first pick a random vertex v. If v is in a large connected component, that is an indication of a small number of connected components.*

Going off of this idea we use the following definition

Definition 6. *For a given vertex $v \in H$ let*

$$\alpha_v = \frac{1}{\textit{size of CC of } v}$$

Fact 7. $\sum_{v \in H} \alpha_v = \#$*connected components in H*

Proof. Suppose there are C connected components and let C_i be the i-th connected component. Using this notation we have:

$$\begin{aligned} \sum_{v \in H} \alpha_v &= \sum_{i=1}^{C} \sum_{v \in C_i} \alpha_v \\ &= \sum_{i=1}^{C} \sum_{v \in C_i} \frac{1}{\text{size of } C_i} \\ &= \sum_{i=1}^{C} 1 \\ &= C \end{aligned}$$

□

We can now finally use this fact to give an algorithm for finding the number of connected components

Algorithm 3.

1. *Pick $k = O\left(\frac{1}{\delta^2}\right)$ vertices $v_1, \ldots, v_k$*
2. *For each $i = 1, \ldots, k$ compute the size of the connected component of v_i*
3. *Output $\frac{n}{k} \sum_{i=1}^{k} \alpha_{v_i}$*

Remark 5. We can compute the number of connected components using either a breadth first search or a depth first search.

Remark 6. This algorithm does have one issue! The size of the connected component of some v_i could be $O(n)$ in which case the algorithm does not have sublinear time. We can remedy this by cutting off the component search once we have checked $O\left(\frac{1}{\delta}\right)$ vertices.

The proof of the validity of this algorithm will be done next class.