6.5240 Sublinear Time Algorithms October 9th, 2024

Lecture 11

Lecturer: Jane Lange *Scribe: Themis Haris*

1 LCA for 3-spanners

In this lecture, we will see a local algorithm for computing a 3-spanner H of a graph G. This algorithm appeared as part of the work of Parter, Rubinfeld, Vakilian and Yodpinyanee [PRVY19].

1.1 Setup and Notation

Definition 1 *Let $G = (V, E)$ be a graph. A subgraph $H = (V, E')$ with $E' \subseteq E$ is a k-**spanner** of G for $k \geq 1$ if for any $u, v \in V$ it holds that*

$$dist_H(u, v) \leq k \cdot dist_H(u, v)$$

*The parameter k is called the **stretch**. An equivalent definition of a k-spanner H is one in which for all $(u, v) \in E \setminus E'$ it holds that*

$$dist_H(u, v) \leq k$$

It is well known that if G has n vertices, then it has a $(2k-1)$-spanner with $O(n^{1+1/k})$ edges. This tradeoff of size and girth might be optimal, if a certain conjecture by Erdős and Gyárfás holds.

For this lecture, let us focus on the case of $k = 2$. We are then seeking 3-spanners of size $O(n^{3/2})$. To add to the difficulty of our problem, we cannot store the whole graph in memory, nor do we have time to process it sequentially. Even if we somehow were able to compute with the whole graph in memory, we are not even able to store or communicate the entire output. Hence, we are going to seek an LCA algorithm to solve the problem.

Recall that in the **Local Computation Model**, our algorithm can *probe* parts of the input graph, and our user can *query* the output. The allowed probes for our setting will be:

- *Neighbor probes* (v, i) get the i-th neighbor of v
- *Degree probes* on v get the degree of v. For this lecture, we will assume an upper bound on the maximum degree as $\Delta \leq n^{3/4}$
- *Adjacency probes* $\langle u, v \rangle$ return i if u, v are adjacent and v is the i-th neighbor of u.

The "user" of the algorithm will then issue *queries* of the form:

Is the edge $(u, v) \in E$ inside the spanner?

We want to be able to answer all such queries in sublinear time – we'll aim for $O(n^{3/4})$ – with respect to a consistent spanner. In a sense, we would like to be able to run queries in parallel if we wanted. To that end, to ensure that consistency is possible, we assume that the computation nodes processing the queries all have access to a common, shared random string that they can read from for free.

1.2 A Global Algorithm

Before we discuss an LCA to solve this problem, let us review a global algorithm for computing 3-spanners that will introduce a key idea.

An influential idea for computing spanners is to partition the graph into *clusters*. Each cluster has a *center*, and every vertex $v \in V$ is at most a small distance away from some center. It turns out we can throw away quite a lot of edges and still maintain this property. We then must make sure that all neighboring clusters are connected to each other somehow. Then, given any two vertices, we can ensure that their distance in H is not blown-up by more than a factor of k by routing the path through the cluster centers. More precisely, consider the following algorithm for computing a 3-spanner H:

1. If a vertex v has $\deg(v) < \sqrt{n}$ (call these vertices *light*), incident edges of v are added to H.
2. Pick a collection of centers S by picking each vertex independently with probability $\Theta(\log n/\sqrt{n})$. In expectation we will have $\mathbb{E}[|S|] = \Theta(\sqrt{n} \cdot \log n)$ centers.
3. If $\deg(v) \geq \sqrt{n}$ (call these vertices *heavy*), assign v to a center $C_v \in S \cap N(v)$, where $N(v)$ is the neighborhood of v. Importantly, note that we must have $S \cap N(v) \neq \emptyset$ with high probability. In fact, we can guarantee that with high probability, for all heavy vertices v, at least one of the first $\sqrt{n}$ neighbors of v will be a center in S (see Lemma 3). Now, we add edge (v, C_v) to H.
4. Each (heavy) vertex v adds exactly one edge to each neighboring cluster (not necessarily to its center though).

Claim 2 *This resulting subgraph H is a 3-spanner.*

Proof If $(u, v) \in E \setminus E'$, then both endpoints are heavy vertices, so we can examine the following two cases:

- If $C_u = C_v$ then $\text{dist}_H(u, v) = 2$ because they are both connected to the same center.
- If u and v belong to different clusters, these clusters will be neighbors. Thus, there must be some w in v's cluster such that $(u, v) \in E'$. Hence, $\text{dist}_H(u, v) = |\langle u, w, C_v, v\rangle| = 3$.

■

Now, we can count how many edges are in H. From step (1), we keep $O(n^{3/2})$ edges incident to light vertices. From step (3) we keep one edge from every heavy vertex, and there's at most $\sqrt{n}$ of them, so that equals $O(\sqrt{n})$ edges. Finally, we add one edge between neighboring clusters for every heavy vertex in each cluster. There are $O(\sqrt{n} \cdot \log n)$ clusters and $O(\sqrt{n})$ heavy vertices, meaning that we can add at most $\widetilde{O}(n)$ edges this way. Overall, the number of edges we add is $O(n^{3/2})$.

1.2.1 Ironing out some details

Lemma 3 *With high probability, for all heavy vertices $v \in V$, at least one of their first $\sqrt{n}$ neighbors lie in S.*

Proof Each neighbor has probability $1 - \frac{\log n}{\sqrt{n}}$ of not being in S. So the probability of all the first $\sqrt{n}$ neighbors to not be selected in S is:

$$\left(1 - \frac{\log n}{\sqrt{n}}\right)^{\sqrt{n}} = \left(1 - \frac{\log n}{\sqrt{n}}\right)^{\frac{\sqrt{n}}{\log n} \cdot \log n} \leq e^{-\log n} = \frac{1}{n}$$

■

1.3 A Local Algorithm

To make our global clustering technique work as an LCA, we first notice that if the query is some edge $(u, v) \in E$ where either endpoint has degree less than $\sqrt{n}$ then we can safely answer **YES**, because all those edges are in the spanner. The problem therefore arises when both endpoints are heavy vertices.

Remark 4 (How is randomness used?) *At this point we should think about how we can have all queries agree on a sampling for S. This is done through the use of the shared randomness that our LCA model has access to. We assume access to a function $f : V \to \{0, 1\}$ drawn uniformly at random based on the random string. Every query computation has $O(1)$ access to this function and this function denotes membership to S. So we see that we can use our access to f to figure out if a vertex is in S or not.*

1.3.1 Attempt 1: Consistency by ordering

Our global algorithm above did not specify which center a heavy vertex should connect to, or which vertex from the neighboring cluster a heavy node should connect to. We need to be specific about these details in the LCA version because we want to ensure consistency. Thus, let us make use of our neighborhood probing ability and declare the following rules:

- A heavy vertex v connects to the **first** vertex $C_v \in N(v) \cap S$.
- A heavy vertex v connects to the **first** neighbor in a neighboring cluster for each such cluster.

With that, suppose we are looking at a query $(u, v) \in E$ where both u and v are heavy. We know that by probing the first $\sqrt{n}$ neighbors of u and v we can find C_u and C_v and thus figure out if u and v are in the same cluster. If they are, we answer **NO**. If not, then the only way $(u, v) \in E'$ is if v is the first neighbor of u in the v's cluster. To check this condition, we can iterate through all $w \in N(u)$ that come before v and check if any of them belong to the same cluster as v. We can check the latter by checking if $C_w = C_v$ in $O(\sqrt{n})$ time: we already have C_v, so we just need to check if the first (out of $\sqrt{n}$) neighbors of w to be in S is C_v. This takes $O(\deg(u) \cdot \sqrt{n})$ time which is not sublinear if $\Delta = \Omega(\sqrt{n})$.

1.3.2 Attempt 2: Multiple centers

The primary wasteful part of our previous approach is that for each $w \in N(u)$ that precedes v we need to find C_w which requires $O(\sqrt{n})$ time. To be more precise, what we need to accelerate is the following membership query:

$$\textit{Is } w \textit{ and } s = C_v \in S \textit{ in the same cluster?}$$

In our previous attempt, whether $s = C_w$ or not was determined by looking at the first $\sqrt{n}$ neighbors of w and seeing if s is the first neighbor of w in S. The key-term here is *first*: we cannot know if s is the first neighbor with this property without looking at all the neighbors.

But what if all the neighbors of a vertex that are in S could be centers? Then we would not have this problem! As long as w and s are connected and s is at most the $\sqrt{n}$-th neighbor of w, they should belong to the same cluster. So, let us redefine our spanner slightly:

- A heavy vertex v connects to all vertices in $C_v := N^{\leq\sqrt{n}}(v) \cap S$, where $N^{\leq\sqrt{n}}(v)$ are the first $\sqrt{n}$ neighbors of v.
- A heavy vertex v **still connects** to the first neighbor in a neighboring cluster for each cluster (this rule remains unchanged).

This adds a few vertices to our spanner, but not too many: Now, each heavy vertex joins $\Theta(\log n)$ clusters in expectation, because we sample with probability $\Theta(\log n/\sqrt{n})$. With this updated definition, our check takes $O(\deg(u))$ time, which is sublinear for $\deg(u) \leq n^{3/4}$.

References

[PRVY19] Merav Parter, Ronitt Rubinfeld, Ali Vakilian, and Anak Yodpinyanee. Local computation algorithms for spanners. *arXiv preprint arXiv:1902.08266*, 2019.