CS 388R: Randomized Algorithms, Fall 2021 November 2, 2021

Lecture 19: Markov chains II; closest pairs

Prof. Eric Price *Scribe: Yeongwoo Hwang and Geoffrey Mon*

NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

1 Overview

In the last lecture we discussed random walks in graphs and began a proof characterizing $C_{u,v}$, the commute time between vertices u and v.

In this lecture finish the proof, give an application to checking $s-t$ connectivity, and begin a unit on computational geometry, starting with the closest pairs problem.

2 Finishing the proof

As a reminder, here is what we are trying to proof.

Lemma 1 (Commute Times)**.** *Consider a random walk on graph $G = (V, E)$. For any two vertices $v, u \in V$, we have the following holds,*

$$C_{u,v} = 2mR_{u,v}$$

$C_{u,v}$ is the commute time and $R_{u,v}$ is the effective resistance between u and v.

Proof. Recall $C_{u,v} = h_{u,v} + h_{v,u}$, where the hitting time $h_{u,v}$ is the time for a random walk beginning at u to reach v. Last class, we showed that if for each vertex v we apply the potential $x_v := h_{v,u}$ then

$$i := \begin{bmatrix} d(v_1) \\ \vdots \\ d(v_n) \end{bmatrix} - 2m\vec{e_u} = L_G x$$

where L_G is the Laplacian for G, $\vec{e_u}$ is the unit vector in the u-th direction, and $d(v)$ is the degree of vertex v. One can easily check that,

$$i = \sum_{v \in V} d(v)(e_v - e_u) \tag{1}$$

Now letting L_G^+ be the pseudo-inverse of L_G, we can solve for the vector x (and thus the hitting times $h_{u,v}$) by

$$x = L_G^+ i$$

However, as we observed last class, this only fixes x up to some additive constant (this is because $[1, \ldots, 1]$ is in the nullspace of L). As a result, we'd like our answer be the *difference* between two

difference x's, such that the constants cancel. To do this, let $(x')_v = h_{v,u'}$ for some other vertex $u' \in V$. Then,

$$\begin{aligned} x - x' &= L_G^+ \left[\sum_{v \in V} d(v) \left((e_v - e_u) - (e_v - e_{u'}) \right) \right] && \text{By eq. (1)} \\ &= L_G^+ \sum_{v \in V} d(v)(e_{u'} - e_u) \\ &= 2mL_G^+(e_{u'} - e_u) \end{aligned}$$

Now, observe that $(x - x')_u = h_{u,u} - h_{u,u'} = -h_{u,u'}$ (recalling we defined $h_{u,u} = 0$), and $(x - x')_{u'} = h_{u',u} - h_{u',u'} = h_{u',u}$. Thus,

$$\begin{aligned} C_{u,u'} &= (x - x')_{u'} - (x - x')_u \\ &= (e_{u'} - e_u)^T (x - x') \\ &= 2m(e_{u'} - e_u)^T L^+ (e_{u'} - e_u) \end{aligned}$$

and since $R_{u',u} = (e_{u'} - e_u)^T L^+ (e_{u'} e_u)$, this is exactly what we wanted to show! □

2.1 The Lollipop Graph

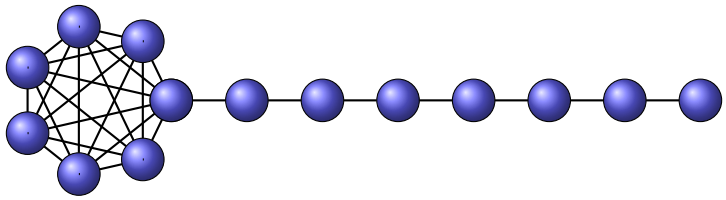

Figure 1: The Lollipop Graph on 14 Vertices

With this in hand, we can go back to our lollipop graph. Since $C_{u,v} = 2mR_{u,v}$, and the effective resistance of the full graph is dominated by the resistance of $\approx n/2$ along the path, we have $C_{u,v} = \Theta(mn) = \Theta(n^3)$. However, this doesn't help us understand $h_{u,v}$ and $h_{v,u}$ individually. In our analysis, we used that $h_{v,u} = (L_G^+ i)_v - (L_G^+ i)_u$, and thus understanding the hitting times requires analyzing the voltage gap between v and u. Let u be some vertex in the clique not on the path and v be the right end of the path. Additionally, let u' be the vertex on both the path and complete graph.

First, for $h_{v,u}$ we are draining $2m$ current from u. Since $\Theta(n^2)$ current is injected into the clique and (u, u') is an edge on the path with resistance 1, we get a voltage gap of at most $O(n^2)$. On the path, we push in $d(v)$ current on each vertex v in the path. Then, between each vertex from v to u', we get a cumulative gain of at least 1 current. Using $V = IR$, we get a voltage gap within a constant factor of $\sum_{i=1}^{n/2} i \in \Theta(n^2)$.

Now for $h_{u,v}$, all of the n^2 current injected into the clique needs to be pulled out from v. Thus, we get a net current of $\Theta(n^2)$ along each of the edges on the path, giving us a $\Theta(n^3)$ voltage gap.

3 Cover times

Define,

$$C_v(G) := \text{Expected time for a random walk of } G, \text{ starting at } v, \text{ to visit all vertices in } G$$

$$C(G) = \max_{v \in V} C_v(G)$$

We can give bounds for some familiar classes of graphs using the tools we've developed so far. For K^n, the **complete graph** on n vertices, the analysis of the coupon collector problem tells us we need $O(n \log n)$ steps to "collect" all the vertices of K^n. For the **path graph**, we know the hitting time between any two vertices is $O(n^2)$. We can bound $C(G)$ by the time it takes to go from an intermediate vertex, to one end of the path, and then to the other end. This is still $O(n^2)$. For the **lollipop graph**, the previous section tells us this is $O(n^3)$. Let's try to find some lower and upper bounds for general graphs.

Lemma 2 ($C(G)$ lower bound)**.** *For a graph G with m edges, $C(G) \geq mR_{max}$.*

Proof. Let (u, v) be the edge which maximizes $R_{u,v}$. We'll denote $R_{\max} := R_{u,v}$. Additionally, note $C_{u,v} = 2mR_{\max} = h_{u,v} + h_{v,u}$. We must have one of $h_{u,v}$ or $h_{v,u}$ is at least $mR_{\max}$. WLOG, assume it is $h_{u,v}$. Then,

$$\begin{aligned} C(G) \geq C_u(G) &\geq h_{u,v} && \text{A spanning walk from } u \text{ must visit } v \\ &\geq mR_{\max} \end{aligned}$$

□

Lemma 3 ($C(G)$ upper bound)**.** *For a graph G with m edges and n vertices, $C(G) \leq 2m(n-1)$.*

Proof. Consider a spanning tree T of G, and a walk through all edges of the tree (which traverses an edge at most twice). This walk clearly visits all vertices of G. Furthermore, we can bound the expected time to perform the walk by the the time to walk each of the $n-1$ edges of T. This is,

$$\begin{aligned} C(G) &\leq \sum_{(u,v) \in T} C_{u,v} \\ &= \sum_{(u,v) \in T} 2mR_{u,v} \\ &\leq 2m(n-1) \max_{(u,v) \in T} R_{u,v} \\ &\leq 2m(n-1) && R_{u,v} \leq 1 \text{ for } (u,v) \in E(G) \end{aligned}$$

□

How good is this bound? On the lollipop and path, the bound appears to match our analysis! However, on K^n, we get $O(n^3)$, rather than the true value of $O(n \log n)$. One culprit is the bound we use on the last line in the above proof; for K^n, $R_{u,v} \approx 1/n$, rather than 1.

Inspired by this, we can get a more precise bound that incorporates $R_{\max}$:

Theorem 4. $C(G) \leq O(mR_{\max} \log n)$

Proof. First, we will show that for an arbitrary vertex u, a random walk starting from any vertex s will reach u after $O(mR_{\max} \log n)$ steps with high probability. To analyze, suppose that we perform $O(\log n)$ *epochs*, each of which consists of $O(mR_{\max})$ steps in the random walk. The expected time to reach u from the starting vertex of the epoch s is

$$h_{s,u} \leq C_{s,u} = 2mR_{s,u} \leq 2mR_{\max}$$

where we make use of the commute time lemma. Then, using Markov's inequality, the probability we need more than $4mR_{\max}$ steps is $\leq 1/2$, and so $4mR_{\max}$ steps will encounter u with probability $\geq 1/2$. Therefore, an epoch of $4mR_{\max}$ steps in the random walk will encounter u with probability $\geq 1/2$.

The probability that all $O(\log n)$ epochs fail to encounter u is $\leq 2^{-O(\log n)}$ which is $\leq 1/n^{10}$ for appropriate choice of constant. Union bounding over all n vertices, the probability that some vertex is not encountered after all $O(mR_{\max} \log n)$ steps is $\leq 1/n^9$, and so the probability that we have covered the entire graph in $T = O(mR_{\max} \log n)$ steps is $\geq 1 - n^{-9}$.

This is enough for us to bound the expected covering time as desired. With probability $1 - n^{-9}$, we will cover the entire graph in the first T steps. With probability $\leq n^{-9}$, we will have some vertices uncovered and will need to keep walking, but $2m(n-1)$ additional steps will suffice in expectation (where we use our earlier upper bound on the covering time starting from any vertex). Hence,

$$\mathbb{E}[\text{covering time}] \leq T + 2m(n-1) \cdot n^{-9} \leq T + 2/n^6 = \boxed{O(mR_{\max} \log n)}$$

where we coarsely bound $m \leq n^2$. □

3.1 Randomized s-t connectivity with small space complexity

To determine whether vertices s and t are connected in an undirected graph, we could use a BFS or DFS deterministically, but this requires $\Theta(n)$ space to keep track of which vertices we have already visited. We can use our covering time result to develop a polytime randomized algorithm that succeeds with high probability and only requires $O(\log n)$ space. Starting from s, perform a random walk of $O(n^3 \log n)$ steps and return that s and t are connected if and only if t is encountered during the random walk. This is because $O(mR_{\max} \log n) \leq O(n^3 \log n)$ (since $m \leq n^2$ and $R_{\max} \leq n$ which is achieved by the two ends of a path), so we have covered all vertices reachable from s with high probability during this walk. We only need $O(\log n)$ space to keep track of the current vertex of the walk.

4 Closest pairs in $\mathbb{R}^2$

Given n points $\mathbf{p}_i = (x_i, y_i) \in \mathbb{R}^2$, how do we find i, j such that the minimum distance between two points σ^* is equal to $\|\mathbf{p}_i - \mathbf{p}_j\|$?

- You can check every pair of points, which would take $\Theta(n^2)$ time.

- There exists a deterministic algorithm that requires $O(n \log n)$ time:

  1. Sort all the points by y-coordinate
  2. Divide the points into two sets of equal size by finding the median x-coordinate, and recurse on each half
  3. Check if there is a closer pair of points that crosses the median line

  Given that the points are sorted, the step of checking for a pair across the median line can be achieved in $O(n)$ time.[1] So, we have $O(n \log n)$ time to sort, and $O(n)$ time for the $O(\log n)$ levels of recursion, which yields a total runtime of $O(n \log n)$.

Today, we will show how to do even better: we can find the closest pair in $O(n)$ expected time using a randomized algorithm. We can begin with a subroutine which provides some intuition. Given a guess σ, we would like to determine whether $\sigma = \sigma^*$, $\sigma > \sigma^*$, or $\sigma < \sigma^*$ holds, without knowing σ^*. To do so, we can imagine some grid of squares, where each square is $\sigma \times \sigma$. We can arbitrarily pick this grid origin to be $(0, 0)$. For each point, we can use constant time to compute the index of its grid square (divide and mod the coordinates with σ) and insert into a hash table that maps grid indices to its contained points.

- If there are > 4 points in a grid square (which we can detect when inserting points into the hash table), then it must be that $\sigma^* < \sigma$ because we can pack 4 points into a $\sigma \times \sigma$ square with pairwise distance $\geq \sigma$ (one point on each corner), but we cannot pack 5 points without some pairwise distance being $< \sigma$. So, when inserting each point, if the target grid square already has 4 points, then we can stop and return that $\sigma^* < \sigma$.
- Hence WLOG, we may assume each square has ≤ 4 points. Then, when inserting the next point, we can check for $\sigma^* = \sigma$ or $\sigma^* < \sigma$ by computing the pairwise distance between the point and all of the points in the surrounding 9 grid squares (there are a total of ≤ 36 points to check, which is a constant, and we can find these points in constant time using the hash table). If neither of these cases ever occur after inserting every point, then $\sigma^* > \sigma$.

It remains to choose σ in order to apply this subroutine. A binary search might not work (e.g. you might not halt because σ is an arbitrary real number). Instead, we can use information from the points that we have inserted before terminating to pick the next σ.

1. Initialize with $\sigma \leftarrow \infty$.
2. Insert each point $\mathbf{p}_i$, checking its distance with all points in the surrounding 9 grid squares to see if it is $< \sigma$.
   (a) If some distance to a neighbor is $< \sigma$, then we update σ to be the shortest distance that we found between $\mathbf{p}_i$ and one of the points that we tested.

[1] Rough intuition: First of all, we can consider just points that are close to the median line (close meaning no further than the closest distance we have found so far). Then, we can divide this area with horizontal lines based on the points to the left of the line, such that each point on the right has its distance computed relative to at most two points on the left.

(b) If we find a new value of σ, stop immediately and restart from the beginning (create a new hash table representing a new grid, and start reinserting all points) with the new value of σ.

3. At the end, return the pair with the shortest distance that we have ever checked.

To show correctness, first note that we always pick σ which is an actual distance between two points, and hence $\sigma^* \leq \sigma$ at all times. In addition, when we restart in step 2 after inserting i points, we have found the shortest distance between two pairs in the first i points (if not, we would have discovered a shorter distance when inserting the previous points, since such a distance would be detected when checking neighboring squares). So, when we terminate, we return a pair with the distance σ^* as desired.

Next, to show the desired time complexity, note that inserting any point takes $O(1)$ time, which includes the hash table operation time as well as the time to check all of the neighboring points (since there are up to $O(1)$ neighboring points, as we will restart the process as soon as there are > 4 points in a grid square since some distance is now $< \sigma$). In addition, if we restart with a new value for σ after inserting i points, then we will never need to restart while reinserting the first i points, because we have guaranteed above that we must have found the pair with shortest distance among the first i points. Hence, handling the ith point requires $O(1)$ time to insert the point and check its neighbors, and $O(i)$ time if we have to restart with a new σ.[2]

$$\mathbb{E}[\text{run time}] \leq \sum_{i \in [n]} (O(1) + \mathbb{P}[\text{inserting } \mathbf{p}_i \text{ forces a restart}] \cdot O(i))$$

By the observations we have made, $\mathbf{p}_i$ forces a restart if and only if the closest pair among $\{\mathbf{p}_1, \ldots, \mathbf{p}_i\}$ includes $\mathbf{p}_i$, so it suffices to bound $\mathbb{P}[\text{closest pair in first } i \text{ points includes } i\text{th point}]$ to be something like $O(1/i)$ in order to achieve $O(n)$ expected run time. We can do this by choosing a uniformly random permutation before running the algorithm to be the order in which we insert points. Then, the probability that one of the points that forms the closest pair in the first i ends up being the ith point is $2/i$. This completes the time analysis.

[2] We are implicitly using time guarantees for the underlying hash table operations e.g. we can use a standard hash table which has expected $O(1)$ time per operation