6.889 Sublinear Time Algorithms December 1, 2020

Lecture 24

Lecturer: Slobodan Mitrović *Scribe: Mihir Singhal*

1 Topics covered

- Massively parallel algorithms for computing random walks

2 Background

Big data has become an increasingly important field in recent years, with total data volume now in the tens of zettabytes. But unprocessed data cannot be used–it is extremely important to be able to actually effectively use huge amounts of data. Several companies have been founded off a single algorithm, such as Akamai and Google. This lecture will cover PageRank, the algorithm which caused Google to start.

3 Problem setting

The main problem we are concerned with is the following: how do you sample a random walk of length L in a small number of parallel steps? The computational model we will using is that of massively parallel computation, which is a good model for real-life data centers. In this model, there are N machines, each with S space. Our graph is stored on all of these machines, so that there is just enough (within a constant factor) space to store all these edges. In other words, the graph has $\Theta(NS)$ edges. The algorithm proceeds in cycles, and each machine can send and receive up to S messages per cycle. We care about the total number of cycles taken, and allow the local running time of each machine to be arbitrarily large.

4 Algorithm idea

The naive algorithm would just sequentially sample each step of the random walk, going to a random neighbor at each step, taking $O(L)$ rounds in total. The question, then, is whether one can do faster than this. The main idea we will use will be to predict where the prefixes of the random walk will end, and prepare their continuations in advance.

The algorithm will be described in three steps: First, we describe an algorithm to sample a random walk on undirected graphs. Then, we use this algorithm to obtain an algorithm for the PageRank problem on directed graphs. Finally, we use the PageRank algorithm to get a random walk sampler on directed graphs.

5 Part I: Random walks in undirected graphs

The main idea of this part is that of stitching. For example, instead of sampling a single random walk of length L, we will instead have each vertex simulate a random walk of length $L/2$, and then stitch together two walks. This will successfully halve the number of cycles needed.

However, when trying to apply this repeatedly by stitching together smaller walks, a problem emerges: due to independence concerns, we cannot use the same walk starting from the same vertex multiple times. Thus, we need a way to estimate how many times the random walk hits each vertex, in order to be able to sample enough walks from each vertex.

We will use the stationary distribution of the random walk to perform this estimation. The *stationary distribution* is defined as the probability distribution on vertices that is fixed by taking one step of the random walk. In matrix notation, letting P be the walk matrix, the stationary distribution is the vector π such that $\pi P = P$. When sampling random walks, after many steps, the number of walks which end at any vertex v is approximately proportional to $\pi(v)$. Due to this property, when computing walks, we can compute a number of walks proportional to $\pi(v)$ starting at each vertex v. This allows us to stitch together many short walks, achieving an efficient random walk algorithm.

For undirected graphs, it is known that the stationary distribution is given by $\pi(v) = \frac{\deg v}{2m}$, where m is the total number of edges and $\deg v$ is the degree of the vertex v. It turns out that the running time of the algorithm ends up being inversely proportional to the minimum element of π. In this case, the minimum element is at least $\frac{1}{2m}$, so the running time is $O(2m)$.

This works well for undirected graphs, since the stationary distribution π is both easy to compute and nicely lower bounded. However, in the case of directed graphs, π can be difficult to compute and have elements as low as $O(1/2^n)$. Thus, something else will be needed for directed graphs.

6 Part II: PageRank in directed graphs

First we will define the PageRank problem. PageRank concerns a walking process which traverses to a random neighbor with probability $1-\epsilon$, and goes to a random neighbor with probability ϵ. This problem is useful for internet search algorithms. Formally, if P is the walk matrix, then the transition matrix of the PageRank algorithm is given by

$$T = (1-\epsilon)P + \frac{\epsilon}{n}\mathbf{1}\mathbf{1}^T.$$

PageRank has the property that the stationary distribution satisfies $\pi(v) \geq \epsilon/n$, which will enable us to use a similar idea again.

It is a theorem of Breyer that PageRank can be approximated by running $O(\log n)$ random walks of length $O((\log n)/\epsilon)$ from each vertex. Because of this, undirected PageRank is easy due to the results of the previous section.

What about directed graphs? Our strategy for directed graphs will be to continuously transform an undirected graph G^U to a directed graph G^D. We will do this in steps, letting

$$G_j = \left(1 - \frac{j}{\log\log n}\right) G^U + \frac{j}{\log\log n} G^D,$$

and transforming a PageRank on G_j to G_{j+1} in steps.

We will design an algorithm that takes a walk w in G_j as input, and which outputs a walk in G_{j+1} or "fail," in a manner such that conditioned on not failing, the algorithm outputs a uniformly random walk in G_{j+1}.

One possible way to do this is to perform rejection sampling: for each edge in the walk, if it comes from G^U (rather than from G^D), then we reject the walk with probability $1/(j+1)$. This does yield the desired distribution on output, and works most of the time when j is large. However, when j is small, this rejects far too often, and as a result we are not able to obtain a walk from G_{j+1} fast enough. In particular, the $j = 0$ case is especially bad: every walk has probability 0 of being accepted.

So, the problem remains how to do this walk conversion for small j. We will use the conversion from G_0 to G_1 as an example to illustrate the method used. The idea will be to pretend that an edge in G^U is actually in G^D. Suppose that we are in the case where each vertex is approximately balanced, i.e., has roughly equal in- and out-degrees. Then, there is a chance we get "lucky": when we want to pick an edge from G^D, the edge that we picked from G^U has roughly a 1/2 chance of going in the right direction. Using this rough idea, we are able to go from G^0 to G^1 when the graph is roughly balanced.

If on the other hand the graph is not balanced, things may go wrong. In the worst case there might be a vertex with in-degree $n-1$ and out-degree 1, and when leaving that vertex there will only be a $1/n$ chance of picking the right edge when walking in G^U. To deal with this, we will transform the graph.

For such a vertex, we will transform it into a series of $\log n$ vertices, so that all the edges that previously entered v now enter v_0, and there are $n/2$ parallel edges from v_0 to v_1, $n/4$ parallel edges from v_1 to v_2, and so on, with the edge that formerly left v now leaving $v_{\log n}$ instead. We may similarly do such a transformation on every vertex. In doing so, each vertex will become approximately balanced, with at least 1/3 of its edges leaving the vertex.

There are some more details to this algorithm which have been omitted, and it turns out that the ϵ in the PageRank will be necessary for this to work.

7 Part III: Random walks in directed graphs

Finally, the PageRank algorithm will allow us to obtain random walks in directed graphs. In particular, we can just use $\epsilon = O(1/L)$, so that the walk is a true random walk with constant probability.