6.895 Randomness and Computation March 15, 2006

Lecture 10

Lecturer: Ronitt Rubinfeld *Scribe: Alexandr Andoni*

This lecture studies some applications of the random walks technique. Specifically, using random walks, we show, in the first part, how to solve s-t connectivity problem in logarithmic space, and, in the second part, we show how to generate uniformly random objects, such as random spanning trees of a graph. The ability to generate objects at random implies the possibility of counting the total number of the respective objects (at least for certain types of objects), as was proved in a previous lecture.

1 Undirected connectivity in logarithmic space via random walks

In the first part of the lecture, we study the problem of s-t connectivity in undirected graph ($USTCON$). This problem takes as input an undirected graph G (e.g., given by an adjacency matrix), and two vertices s and t. The algorithm has to answer *Yes* if there is a path from s to t, and *No* otherwise. Note that $USTCON$ can be easily solved in linear time (for example, by a Breadth First Search). However, here, we are interested in small space (logarithmic), and a BFS approach takes super-logarithmic amount of additional space (the space to store a "visited" bit for each node).

We investigate whether we can solve $USTCON$ using only $O(\log n)$ space (we do not count the input of the problem towards the space usage of the algorithm). Below we prove that we can solve $USTCON$ using $O(\log n)$ space, but in a randomized sense: the algorithm will sometimes fail to give the right answer. We formalize these notions below.

Definition 1 *Let* **L** *be the class of languages that can be decided by a Turing machine that has read-only access to the input, as well as read/write access to* $O(\log n)$ *bits of additional storage.*

Next, we define **RL**, the one-sided error variant of **L**.

Definition 2 *Let* **RL** *be the class of languages* L *for which there exists a Turing machine* M_L *such that:*

- M_L *has read-only access to the input, as well as read/write access to* $O(\log n)$ *bits of additional storage;*
- *On input* x*, if* $x \in L$*, then* $Pr[M_L(x) = accept] \geq 2/3$*;*
- *On input* x*, if* $x \notin L$*, then* $Pr[M_L(x) = accept] = 0$*;*
- *On any input* x*,* $M_L(x)$ *runs in polynomial time.*

In this lecture, we prove that $USTCON \in$ **RL**, but, recently, it has been proven in [Rei05] that $USTCON \in$ **L**. We might see a sketch of the Reingold's result later in the class.

The algorithm $USTCON$ is very simple and consists of walking randomly on the graph until we hit t. More precisely, the algorithm is

1. Start at the vertex s.

2. Take a random walk for N steps (at each step we choose a random neighbor of the current node).

3. If we visited t at least once, output Yes.

4. If we never visited t, output No.

Next we prove that we can implement the algorithm in logarithmic space, and that $N = 4n^3$ suffices. The space usage is comprised of storing the current and the next pointer ($O(\log n)$ bits), and the length of the walk ($O(\log N) = O(\log n)$ bits).

For correctness, note that if s and t are not connected, then $\Pr[Yes] = 0$. If s and t are connected than we can prove that, with probability at least $2/3$, we hit t at least once. The argument comes from the cover time bound deduced in the previous lecture:

$$\begin{aligned}
\Pr[\text{walk for } 3n^3 \text{ steps and don't see } t] &= \Pr[\text{the length of the random walk from } s \text{ to } t > 3n^3] \\
&\le \frac{E[\text{length of the random walk from } s \text{ to } t]}{3n^3} \\
&\le \frac{E[\text{cover time}]}{3n^3} \\
&\le \frac{n^3}{3n^3} \\
&= \frac{1}{3}
\end{aligned}$$

The first inequality uses Markov's inequality, and the last inequality uses the fact that the expected cover time is at most n^3 in any graph.

Question: what if we start at a random node (and check, e.g., if we hit s and t)?

Answer: We still need a $\Omega(n^3)$-length walk since there are graphs where for most nodes (e.g., fraction of $> 99/100$), the expected time of the walk from that node to t is $\Omega(n^3)$ (think of lollipop graph with the "clique" component 100 times bigger than the "line" component).

Note 3 *Observe that* **L** *is in* **P***, i.e., any deterministic algorithm using* $O(\log n)$ *additional space runs in polynomial time (since there are only* $O(1) \cdot 2^{O(\log n)}$ *different states of the algorithm). Also* **RL** *is in* **P** *by the definition of* **RL***.*

A big open question is whether $\mathbf{L} = \mathbf{RL}$ or $\mathbf{L} \neq \mathbf{RL}$. The result of [Rei05] is a suggestion in the positive direction (this result proves that "$\mathbf{L} = \mathbf{SL}$"); for some more recent progress on **RL** vs. **L** question, see, e.g., the paper of Reingold, Trevisan, and Vadhan [RTV06].

2 Random Walks and Rapidly Mixing Markov Chains

In the second part of the lecture, we show how the technique of random walks helps in uniform generation of random objects. Examples of such "random objects" are: a random perfect matching in a given graph G, a random spanning tree in a given graph, etc. Note that for both of these examples we can find one object in polynomial time, but it is unclear apriori how to generate one at random. This task seems especially hard given that, in general, there could be an exponential number of these objects (e.g., there is an exponential number of spanning trees in a complete graph). We will show below a general framework for generating such random objects.

For uniform generation of these random objects, we use the following general framework:

1. Generate any solution, not necessarily a random one (e.g., find some spanning tree of the graph).

2. Modify locally the solution for N times; each local modification is random (e.g., modify spanning tree by choosing some edge at random, put the edge into the spanning tree, and remove some existing spanning tree edge to preserve the tree structure).

3. After these N steps, we get a uniformly random object (to be more precise, we obtain an approximately random object, in a sense that will be defined later).

Note 4 *Note that approximate uniform generation implies we can approximately count the number of the objects. Thus, e.g., we can count the number of perfect matchings in a given graph, number of spanning trees, etc. Approximate counting of such objects has implications in physics, specifically, about the "properties/energies of the system".*

This framework is best viewed as performing a random walk on a Markov Chain. Specifically, we model the problem of the uniform generation with the following Markov Chain (MC):

- A state of the MC corresponds to one object (e.g., one state corresponds to one spanning tree). Another reasonable option is to have several states per one object (e.g., several states correspond to the same spanning tree, but for each spanning tree there is at least one corresponding state). We can use the acceptance/rejection principle for correcting the probabilities of generating the objects (discussed within uniform generation in a earlier lecture).
- Edges of the MC correspond to the "local changes" (e.g., all edges leaving a state correspond to the "local changes" we can perform on the corresponding spanning tree). We should be able to compute a "local change" in time that is polynomial in the size of the object.
- Stationary distribution of the MC should be (ideally) equally distributed amongst all the states (thus, roughly, uniform over the objects).
- Time to reach the stationary distribution (or a distribution close to it) should be polynomial in the size of the object (e.g., polynomial in the size of the graph in the case of uniform generation of spanning trees). This usually means that we need to be logarithmic in the size of the MC (note that MC is usually exponential in the size of the object).

Below, we use the following notation:

- A is the MC (A_{ij} is the probability that we transition to state j, given that we are in state i).
- π is the current distribution over the states of the MC;
- $\pi^{(0)}$ is the starting distribution (e.g., $\pi^{(0)}(\text{start state}) = 1$ and zero elsewhere);
- $\pi^{(0)}A^t = \pi$ is also the current distribution;
- $\tilde{\pi}$ is the stationary distribution of the MC: $\tilde{\pi} = \lim_{t\to\infty} \pi^{(0)}A^t$;
- l_1 distance of two distributions x, y is $l_1(x,y) = \|x-y\|_1 = \sum_i |x_i - y_i|$.

Definition 5 *For $\varepsilon > 0$, the* mixing time *of the Markov Chain A, denoted by $T(\varepsilon)$, is defined as the minimum integer t such that for any $\pi^{(0)}$, we have that $\|\tilde{\pi} - \pi^{(0)}A^t\|_1 < \varepsilon$.*

Definition 6 *The MC A is* rapidly mixing *if $T(\varepsilon) = poly(\log|V|, \log\frac{1}{\varepsilon})$, where $|V|$ is the number of states of the MC.*

2.1 Generating random spanning trees in a graph

Next, we materialize the framework for a particular problem. Specifically, we concentrate on generating spanning trees (approximately) uniformly at random. Note: there is a known deterministic algorithm for counting the number of spanning trees, so this method is not necessarily "the best", but this is not the case for all objects we can generate via this framework. We study spanning trees to only illustrate the power of the framework.

We construct the following MC for the generation of spanning trees:

- Start with some spanning tree T, and orient all edges towards the some vertex x that we call the root;
- The states of the MC are rooted directed spanning trees (the edges always point towards the root);
- A step would be as follows:
  - Choose some edge (x, y) that is incident to the root x (in the graph G).
  - Insert the edge (x, y) (in this orientation) in the spanning tree T.
  - Remove the edge outgoing from y.
  - y becomes the new root.
- At the end of the random walk in the MC, we output the current spanning tree T (without the orientation of the edges).

Below, we present an example of a state, and the steps that can be taken from this state. Figure 1 represents a state with the corresponding spanning tree (rooted and directed spanning tree). The edges marked 1, 2, and 3 are the edges that can be chosen for the next step. Figure 2 represents the spanning trees resulting from taking each of these steps.

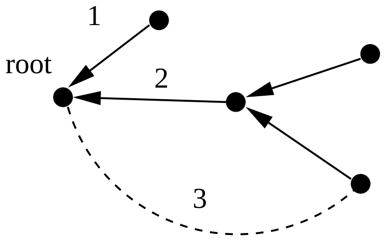

Figure 1: An example of a state, with the corresponding spanning tree. From this state there are three possible steps: one corresponding to each of the edges labelled 1, 2, and 3. The dashed edge is an edge present in the graph, but not part of the spanning tree.

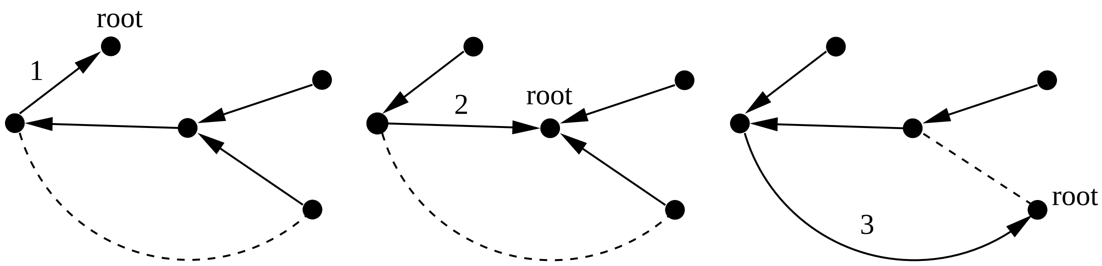

Figure 2: States resulting after taking a step corresponding to choosing one of the edges 1, 2, and 3 respectively. Note that only for edge 3 we do change the actual tree structure of the spanning tree.

We can prove for this MC that it is rapidly mixing and that the output spanning tree is almost uniformly random. Specifically, note that the outdegree and indegree of a state are both equal to the degree of the root of the corresponding spanning tree. This observation lets us prove the following lemma:

Lemma 7 *The stationary distribution of the above MC is* $\tilde{\pi} = \left(\frac{degree(root\ of\ state)}{Z}\right)_{state}$, *where* $Z = \sum_{states} degree(root\ of\ state)$.

Proof Next time.

Further, note that each possible spanning tree corresponds to n states (since there are n possible different roots in the spanning tree). Thus, the stationary distribution is roughly what we need (though, we need to adjust the probabilities as will be described in the future lecture).

The main technical part is to show that the MC is rapidly mixing. For this purpose we will use the coupling lemma.

References

[Rei05] O. Reingold. Undirected ST-connectivity in Log-Space. *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, 2005.

[RTV06] O. Reingold, L. Trevisan, and S. Vadhan. Pseudorandom Walks on Regular Digraphs and the RL vs. L Problem. *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, 2006.