6.842 Randomness and Computation	March 31, 2020
Lecture 12	
Lecturer: Ronitt Rubinfeld	*Scribe: Jonathan Shoemaker*

1 Undirected S-T Connectivity in Deterministic Log-space

Given: undirected graph G with nodes s, t
Question: are s, t in the same component?

Last time we saw a randomized algorithm to solve this problem. We want to de-randomize this. However, it seemed difficult to de-randomize the algorithm. This is because the randomized algorithm started at s, took steps for a while, and then we argued it should have gotten to t if they were in the same component. De-randomizing these randomized choices seemed hard because we did not even store them as we went: we simply stored the current node and the number of steps we had taken.

2 Expanders

First we will focus on a case where it is easy to de-randomize the connecitivy algorithm. This case is called an "expander."

Definition 1 (N, D, λ)-graph
N is the number of nodes in the graph. D is the max degree of the graph. λ is an upper-bound on λ_2, the second largest eigenvalue of the transition matrix of the graph (this is often associated with the connectivity of the graph).

Well known fact (due to Tanner, Alon-Milman):
$\forall\, \lambda < 1$, $\exists$ some $\varepsilon > 0$ such that, $\forall$ (N, D, λ)-graphs and $\forall\, S$ such that $|S| < N/2$, we have that $|N(S)| \geq (1 + \varepsilon)|S|$, where $N(S)$ includes the vertices of S and all vertices connected to a vertex in S.

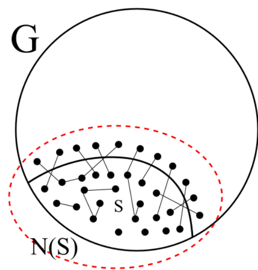

Figure 1: Schematic of S and $N(S)$. $N(S)$ is contains all nodes within the red, dotted oval

- there are edges completely within S (pictured) and edges completely outside of S (not pictured)
- there are also edges that "cross" S
- $N(S)$ is the sum of nodes within S and the nodes connected by edges that cross S

Now, we consider why we care about having expanders. If $|N(S) \setminus S| \geq \varepsilon \cdot |S|$ then a lot of new nodes are added when expand from S to $N(S)$. If you consider starting at one node and growing the component, the component grows with exponential growth.

What is the diameter of an (N, D, λ)-graph with $\lambda < 1$? Consider the path between two nodes. If we expand from the two nodes then the sets formed will eventually intersect. Since we know that the neighborhoods grow exponentially, this means the path between any pairs of nodes must be logarithmic. Thus, the diameter is $O(\log n)$.

2.1 Low Diameter Solution

Now, we consider why it is easier to compute connectivity when the graph is an expander rather than just a general graph. Actually, we are going to have an algorithm that will work whenever the diameter is logarithmic.

Idea for "low diameter" (meaning each component of the graphs has diameter of length $O(\log n)$), constant degree graphs:

- start at s
- enumerate all paths of length $O(\log n) = l$. The number of these paths is $D^l = D^{O(\log n)} = N^{O(1)}$ because D is the constant degree of the graph
- if we ever see t, output "connected", otherwise output "disconnected"

Is this algorithm correct? Yes, if s and t are connected then the path between them is at most length l and thus we will eventually see t.

What are the space requirements? The enumeration is basically just a DFS. So we essentially need to keep track of a DFS in logspace. The way we would normally do this requires making our DFS calls in a stack and takes a lot of space. However, we only need a constant number of bits for each step. This is because we can just store which of the D edges we took from a node and D is constant. Further, we only need to store $O(\log n)$ length paths at a time. Thus the total storage for keeping track of the DFS is just $O(\log n)$. We note that the runtime of the algorithm might not be stunning (meaning something like $O(n)$). However, this does not really matter because we only care about running in logspace (also all logspace algorithms will have polynomial runtimes).

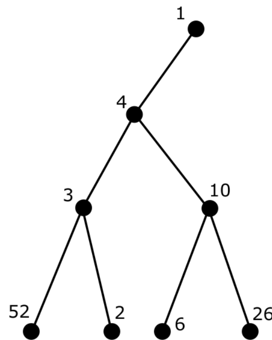

Figure 2: An Example Graph (happens to be a tree)

On the left is an example graph (which is a tree just for ease of example). A potential DFS order of the graph would be $1 \to 4 \to 3 \to 52 \to 3 \to 2 \to 3 \to 4 \to 10 \to 6 \to 10 \to 26 \to 10 \to 4 \to 1$.

Normally, a DFS would store an entire stack containing each node on the path. However, we can just store the path with a base D number of the appropriate length. If we need the path at any point we can recreate it by walking through the base D number. This is done particularly quickly but still remains in logspace. A concrete example of what this might look like is with binary trees. The base D number would just be a sequence of L's and R's denoting whether we went down the left child or the right child. Reconstructing the path is pretty straightforward in this case.

Now, we look at the total needs for the algorithm. We need $O(\log n)$ to store the start node. Then we also need $O(\log n) \cdot O(1)$ to store the path (the constant factor is due to D, the constant degree). Again, using this to find previous nodes on paths and the like makes the runtime worse but it does not matter for our purposes.

3 General Graphs

We still have a problem: not all graphs are like this (meaning "expanders"). The three properties we relied on were not always present:

- (N, D, λ) for $\lambda < 1$
- $O(\log N)$ diameter
- constant degree

There is a theorem for general graphs that is useful.

Theorem 2 $\forall$ *connected, non-bipartite graphs* G, $\lambda_2(G) \leq 1 - \frac{1}{DN^2}$

The theorem gives us a λ that is strictly less than 1. However, the dependence on N makes this not particularly helpful (because the expansion does not give us the same logarithmic diameter properties as before).

An idea that might seem helpful is taking powers of the matrix. If G is (N, D, λ) then G^t is (N, D^t, λ^t). The powers retain the same answer as the original graph - connectivity does not change. Further, the powers reduce λ_2 which is the entire flaw with general graphs. However, the power also increase D. This is not good since we need D to be a constant.

Despite the degree increase, powering still seems incredibly helpful. We will thus use powering but will add in another operation that reduces the degree without increasing λ_2 too much.

3.1 Base Graph

Now, we define what is called a "Base Graph," and show one that exists.

Theorem 3 $\exists$ *a constant* D_e *and a* $((D_e)^{16}, D_e, 1/2)$*-graph*

The graph above is important for a few reasons:

- the graph has a constant size and a small λ_2
- the graph can be generated and used regardless of what input we have (because it is constant sized)
- we can always find the graph by enumeration in logspace (because checking constants is easy)

Essentially, we can now find small expanders which will prove to be pretty helpful

3.2 Making degree constant

We want to be able to assume that G has constant degree. We can transform G to try to achieve this. One way of doing this is to turn in each node into a cycle consisting of nodes equal to the original number of edges connected to the node. Then, the edges connect the corresponding nodes in the cycles of the vertices the edges originally connected.

In Figure 3 the left is the original graph. It has degree $\leq N$ and number of nodes equal to N. The right is the cycle-replaced graph. It has degree ≤ 3 and number of nodes $\leq N^2$. We only do this replacement once though and when we take logs having a square does not make a difference. Further, the new graph retains all of the connectivity properties of the original graph (the answer for *s*-*t* connectivity is always the same) and has constant degree.

This leads to the idea that maybe we want to power the graph then use the cycle expansion to decrease degree then power again, and so on. The issue with this is that the cycle transformation affect λ_2 too negatively for this to work. We will want to do something very similar to this except replace nodes with the base graph instead of the cycle (it will not hurt our λ as much because it is an expander).

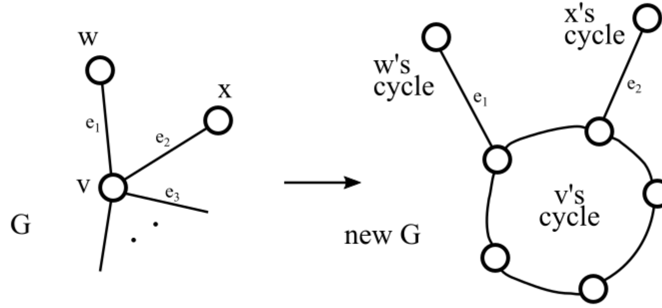

Figure 3: Example of cycle replacement decreasing degree

3.3 Rotation Maps

We need to find some way of representing the graphs such that we can access them in logspace. To this we will define a "rotation map."

Definition 4 Rotation Map
$Rot_g : [N] \times [D] \to [N] \times [D]$. $Rot_g : (v, i) \to (w, j)$ if the ith edge of v leads to w and the jth edge of w leads to v.

The rotation map allows us to go back and forth across the same edge very easily and is helpful because for each node, the map gives the node info on how all its neighbors point back to it.

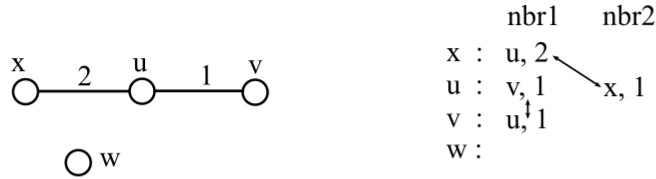

Figure 4: Example rotation map. The graph is on the left and on the right is the table storing connections for each node

3.4 Replacement Product

Now, we will discuss what is called the replacement product. It generalizes the cycle replacement we did earlier. To do the replacement product $G \circledR H$ we are given two graphs: G, which is D-regular and has N nodes, and H which is d-regular and has D nodes. We then create a graph G' which has $N \cdot D$ nodes and degree $d + 1$ ($d + 1 << D$).

We replace all of the nodes $v \in G$ which a copy of h. Further, we will add in the H edges such that each H_v has is isomorphic to H in both nodes and edges.

An edge that is originally between u and v is added between a node of H_u and a node of H_v. Specifically, if u is the ith neighbor of v and v is the jth neighbor of u (in G), then we add an edge from the ith node of H_v to the jth node of H_u.

Each node in the replaced graph will only have 1 edge added from the original graph. In addition to the d edges from the H_v this gives us the degree of $d + 1$.

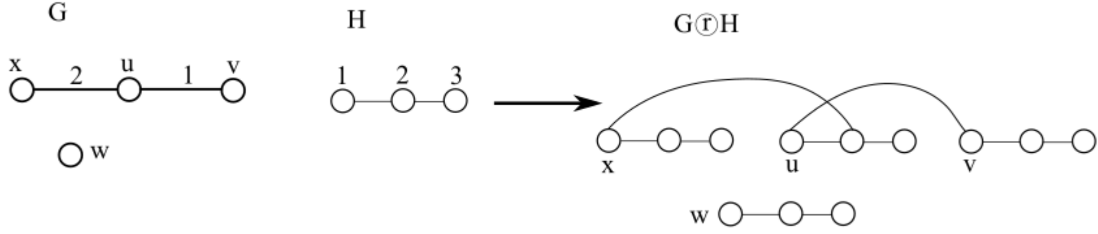

Figure 5: Example of replacement product

3.5 Zig Zag Product

Now, we define the zig-zag product, which is similar to the replacement product. Once again to do the zig zag product $G \textcircled{z} H$ we are given two graph G and H with the same constraints as in the replacement product. Then we will create G'' which will have $N \cdot D$ nodes and degree d^2 (so the degree is a larger constant than in the replacement product).

The nodes are similar to in G': each v in G is replaced by a copy of H. The edges are a pit different. They will correspond to paths of length 3 in G'. We mention that we will also call H_i the "cloud" of i. $(u, v) \in G''$ if and only if $u \in H_i$, $\exists w \in H_i$ such that $(u, w) \in E(H_i)$, and $\exists z \in H_j$ such that $(w, z) \in G \textcircled{r} H$ and $(z, v) \in E(H_j)$ (so v is also in H_j). Essentially what edges correspond to is crossing an edge in the expansion of a node then taking an original edge of G and then crossing another edge in the expansion of a node.

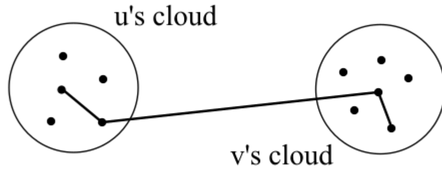

Figure 6: Example of added edge in zig zag product

- there are d options for the edge within H_u
- there is just one option for the edge that crosses clouds
- there are d options for the edge within H_v
- in total, this gives us d^2 edges corresponding to each original edge and thus a new degree of d^2

When we take a zig-zag product, the minimum cut in the new graph will be similar to the cut size in the original graph (in a relative sense). This means it does not ruin λ_2 and we know it still decreases degree to a constant.

The clouds are expanders. We want the whole cloud to go to one side of the cut and only edges that were in the original G to cross the cut. We want walks on H to not really do much but walks on G to contribute a lot (this allows us to preserve λ_2).

Theorem 5 *For $\alpha \leq \frac{1}{2}$, G an (N, D, λ)-graph and H a (D, d, α)-graph (e.g. the base graph), we have that $G \textcircled{z} H$ is an $(ND, d^2, \lambda_{G \textcircled{z} H})$-graph such that $\frac{1}{2}(1 - \alpha^2)(1 - \lambda) \leq 1 - \lambda_{G \textcircled{z} H}$*

The important consequence of the above theorem is that λ in the zig-zag product is a constant below 1 - this is what we need to run our expander graph algorithm. Formally, we have that $\lambda_{G \textcircled{z} H} \leq 1 - \frac{1}{2}(1 - \alpha^2)(1 - \lambda)$. Then, because $\alpha \leq \frac{1}{2}$, we have that this is at most $1 - \frac{3}{8}(1 - \lambda)$ which is at most $\frac{2}{3} + \frac{\lambda}{3}$. Thus, since $\lambda < 1$ we have that $\lambda_{G \textcircled{z} H} < 1$.

3.6 Full Algorithm

We now want to use the transformations we have found important: taking the power of the graph and doing the zig-zag product.

We are given a graph G that is D^{16}-regular on N nodes and a graph H that is D-regular on D^{16} nodes (if we had a big G we could just do one replacement at the start).

The transformation will be as follows:

- $l \leftarrow$ smallest integer such that $(1 - \frac{1}{DN^2})^{2^l} < \frac{1}{2}$
- $G_0 \leftarrow G$
- $G_i \leftarrow (G_{i-1} ⓩ H)^8$

We will then output G_l.

The idea is that we are constantly maintaining constant degree with the zig zag product and we are constantly decreasing λ_2 with the powers. We pick l so that we do this enough that we get an expander.

Formally, G_l has some nice properties. We have that the number of nodes is $N \cdot (D^{16})^l$ which is just polynomial in N. Further, you can show that $\lambda(G_l) \leq \frac{1}{2}$.

The zig-zag product does not ruin λ but decreases the degree. We just need to pre-process the graph G (we can just do the cycle replacement once) then we do the above steps to find G_l. Then we just run the algorithm for expanders on G_l. We have always guaranteed that connectivity did not change with either the power or the zig zag product.

All that is left is to make sure everything can be done in log space. Powering is in log-space (it is essentially 2-step paths). Further, zig-zag is just 3-step walks and thus in log-space as well. Details are omitted on how exactly to maintain everything (such as how we need to use rotation maps).