In the last class we went over Network Coding. At a high level, this is a coding scheme where nodes perform some computation on messages they own before transmitting new messages to their neighbors. Network coding exploits the fact that information can be combined together. This idea is what allows us to maximize the information throughput of the network. Network coding is robust against link failures. In the case of a dynamic network, you might not know which links are up and which links are down. If you use some precomputed static routing strategy and links go down, you might not be achieving information throughput. In Network Coding, however, this doesn’t matter at all.

Now let’s actually work with a concrete type of network coding - Random Linear Network Coding - and prove some results about it.

1 Random Linear Network Coding

Suppose we are trying to send \( k \) messages, \( m_1, m_2, \ldots, m_k \) in \( \mathbb{F}_q^* \). We are given \( G = (V, E) \), where \( G \) is a directed acyclic graph. Suppose we are playing a one-shot game on this network - that is, we can only transmit a single packet over each edge (so whatever packet we transmit over that edge had better be useful!) Suppose we have \( k \) incoming links to the source. The strategy we use is as follows: each node sends a packet across an edge which consists of a random linear combination of the messages it has. That is, the message sent from the source is \( m = \sum_{i=1}^{k} \alpha_i m_i \), where the source selects a different set of \( \alpha \)'s for each edge. The form of the message we send is \((\alpha, m)\) - that is we send along both the coefficients and linear combination as our message.

An intermediate node receives some \( d \) messages \( m'_1, m'_2, \ldots, m'_d \) where \( m'_i = (\alpha_1, \ldots, \alpha_k, \sum_{i=1}^{k} \alpha_i m_i) \) and just linearly combines them, which produces a new linear combination of our \( m_i \)'s (by linearity).

If you have \( k \) input messages, you need at least \( k \) linearly independent equations over the field in order to recover the \( m_i \)'s. We have:

\[
p_i = \sum_{i=1}^{k} \alpha_i m_i
\]

To recover \( m_i \), we just apply gaussian elimination on the coefficients. As an aside, note that if \( k > \mincut(G) \) there’s no hope of getting our \( k \) packets across. From the source’s perspective, you get some mixture of the \( k \) messages, but these will be dependent, which disallows us from retrieving the original \( k \)'s.
Another aside: If for some reason you want each vertex to send the same lc over all of its out-edges, you need to use the vertex min-cut instead of the edge-min cut.

**Theorem 1** If for every sink in a DAG network the mincut between the source and the sink is at least $k$, then Random Linear Network Coding with $k$ messages will succeed whp if field size $q$ is sufficiently large.

**Proof** Consider some arbitrary sink. There is a mincut of size at least $k$ between the source and sink by assumption. Therefore, consider the max-flow corresponding to this cut. As we have integral edge weights, we have an integral flow, and by the definition of a flow, have at least $k$ edge-disjoint paths between the source and sink.

Consider the following sequence of cuts. The first cut contains solely the source. We add vertices to the cut in topological order, from the source. There are $O(n)$ such cuts - each time we shift the highest node not in the cut, into the cut. Define ‘red edges’ to be edges in the max-flow between the source and sink (edges on the disjoint paths in the max flow). Consider the following event: $(B_i : \text{red edges crossing cut } i - 1 \text{ are full rank, but red edges crossing cut } i \text{ are not}).$

If the vertex we’re adding to the cut has just one red edge, the probability that we end up picking a bad coefficient is just $\frac{1}{q}$. Suppose that we have two-red edges. The failure probability for the first edge is now $\frac{1}{q^2}$. For the second edge, it now fails with probability $\frac{1}{q^2}$. One way to think about this is that we’re measuring the probability that we lose a dimension when we add back edge $1, 2, \ldots, i$. Losing a dimension on the first edge corresponds to us picking some $d \ b$’s s.t. the resulting linear combination has lost a dimension - each choice happens with probability $\frac{1}{q^2}$. Extending this idea, we have:

$$Pr(B_i) = 1 + \frac{1}{q} + \frac{1}{q^2} + \ldots + \frac{1}{q^d} < \frac{2}{q}$$

which is the failure probability for a single cut failing. The probability that all of them fail by the union bound is $\leq \frac{2n}{q}$. That is:

$$Pr(\geq 1 \text{ failures}) \leq \frac{2n}{q}$$

Looking at this, clearly $q$ has to be a lot bigger than $n$ for this to have any chance of succeeding. The last step in the proof is to extend from one sink, to work over all sinks. This is just another union bound over all the sinks, which gives us a final probability of $\frac{2n^2}{q}$ (the probability that at least one fails). The probability of success is therefore $1 - \frac{2n^2}{q}$ (no failures).

Next time, we’ll see a proof that we can drop the DAG assumption, and get that $q = 2$ will work really well! Note that $q = 2$ would completely break the proof given here, so we’ll have to use a new proof technique for next time. ■
2 Project

Try to find a problem that is somehow related to randomization. Hopefully interesting enough that you’re willing to look at it for the next three months. No requirement to make something publication worthy, it might! Try to pick something that with some luck has a chance to become a paper. Pick groups of at least size 2. There’s a website update on the project side with a bunch of project ideas, so if you don’t have an idea of what you want to work on take a look there.