CPSC 320 Notes: Pipes, Knives, and Randomization

April 8, 2015

Let's have some quick randomization fun with a network flow kicker!
But first, an interesting note: $\lim_{n\to\infty}(1-\frac{1}{n})^n = \frac{1}{e}$.

1 A First (Foolish-But-Maybe-Effective?) Randomized Algorithm

You want to find the largest number in an array of n distinct integers. You cannot **imagine** any algorithm that will work.

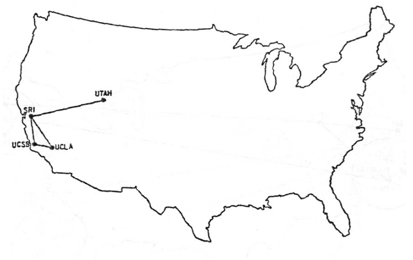
1. Your first stab: Pick an entry (uniformly) at random and report that back. What is the probability that you report an **incorrect** answer? What is the probability asymptotically, in terms of n?

2. Your second stab: Try n times to pick an entry at random, keeping track of the largest you've seen so far. Report that largest number back. What is the probability that you report an **incorrect** answer? Asymptotically?

3. How many times should you run your algorithm from the previous part—keeping track of the largest answer so far—so that the probability of getting an **incorrect** answer is $\frac{1}{n}$.

2 A Day Without Internet (Dum-Dum-Dum!)

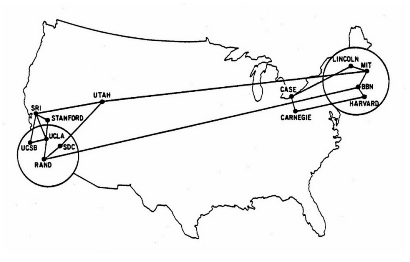
1. Here was the Internet (actually, ARPANET, its predecessor) as of December 1969:

   The ARPANET in December 1969

   How many wires would need to be cut to partition the Dec 1969 ARPAnet?

2. How about the number of wires to partition the Internet of 1970?

3. Rewrite these questions as a general, algorithmic problem. What are the inputs? What should the output be? What constraints are there on input and output?

3 That Chapter We Skipped

Here's a problem called one of "min-cut", "max-flow", or "network flow": Given a non-negative-weighted, directed graph $G = (V, E)$ and two nodes $s, t \in V$, where s is the "source" and t is the "sink", consider each edge to be a "pipe" that can carry as much "flow" or "current" as its weight and determine how much total flow can pass from the source to the sink. (It's a sewer system, or a set of train tracks, or an Internet network, or a **huge** number of other things!)

There's a well-known polynomial-time solution to this problem.

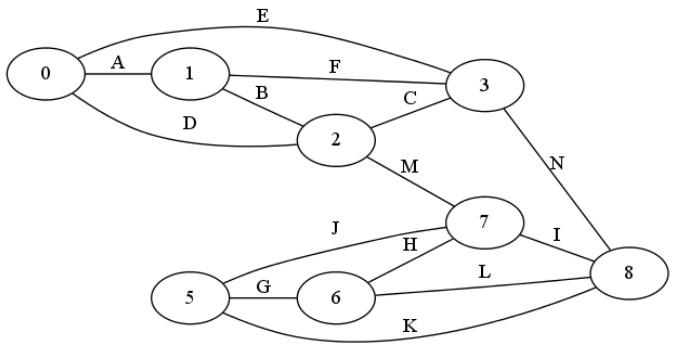
How can we **use** this to solve our problem in polynomial time? (I.e., give a reduction from our problem to this network flow problem, ensuring that you make no more than a polynomial number of "calls" to the network flow problem. You will want more than one, however!)

4 A Randomized Approach to Undirected Min-Cut

1. Remember "edge contraction"? We'll redefine it very slightly: When we contract an edge in a graph, we create a new graph identical to the old one except that the two nodes x and y incident on that edge have been merged into a single node x/y. All edges between x and y are deleted; all other edges previously incident on x or y are now incident on x/y instead. (That means a pair of vertices may have multiple edges between them; we'll allow that!)

   Contract the edge (SRI, STANFORD) in the 1970 APRANET graph above and draw the resulting graph. (No need to draw the United States!)

2. Consider the following undirected graph:



   This is drawn in such a way that we can easily find the minimum cut-set (set of edges that, if removed, partitions the graph), but even this small graph would be tricky to analyse if drawn differently.

   Run though this extraordinarily naïve algorithm once and write down the letters of the edges you find:

```
While |V| > 2:
  Choose an edge (u, v) uniformly at random from among all edges
  Contract (u, v)
Report the set of edges that remain as the cut-set
```

   You'll need a way to choose which edge to contract next. Find your own way to choose randomly or use the (relevant) letters in order from this random.org query.

3. **Upper-bound** the probability that this algorithm contracts one of the edges that **is** in the minimum cut-set on its very first iteration. (*Hint:* let the size of that cut-set be k. What the minimum degree of any node in the graph?)

4. **Upper-bound** the probability that—having avoided contracting any edge in the minimum cut-set so far—the algorithm then contracts such an edge on its j-th iteration (where the first iteration above was the 0-th).

5. Overall, what is the probability that this naïve algorithm returns a correct result?

6. Let $x = \frac{n(n-1)}{2}$. What is the probability that the algorithm returns an **incorrect** result in terms of x? (Use your previous answer!)

7. If we run the algorithm x times, what is the probability that it returns an **incorrect** result? Let's call this the "full naïve algorithm".

8. How many times must we run the "full naïve algorithm" so that the probability of an incorrect result is $O(\frac{1}{n})$?

   This is **not** yet as efficient as the network flow solution. However, it turns out that the odds of contracting "the wrong edge" go up as we get close to the end of the naïve algorithm. Thus, we can improve it by cutting it off just before the end and deterministically solving the small remaining graph (e.g., with network flow) and also by using divide-and-conquer to chop the iterations into stages and recursively trying everything after a given stage twice, thus doing exponentially more runs of the later stages than the earlier ones.

5 Challenges

1. Imagine that you have some constant, non-zero probability $0 < p < 1$ of producing an optimal answer to a problem using some randomized process. Each time you run that process, the probability of an optimal result is independent of all prior runs. You can tell "better" results from "worse" results. Give an algorithm that produces an **incorrect** answer with probability $O(\frac{1}{n})$.

2. Continuing the previous problem: Give an algorithm that produces an **incorrect** answer with probability $O(\frac{1}{2^n})$.

3. There is a strange, dark cave with a long entrance passage that then branches into two paths. The paths curve back together and dead-end at a 1 meter wide apparently impassable wall of rock, but in fact—I claim—the wall is a magic door, and only I know the password to make it open and close.

   You don't believe me, and I refuse to tell you the password so you can check for yourself. How can I efficiently prove to you (with tremendously high probability) that I **do** know the magic password without telling you what it is?

   (See: How to Explain Zero-Knowledge Protocols to Your Children. Protocols like this can, for example, give you a voting system where voters can check that their votes were counted correctly while being unable to prove how they voted (and therefore be coerced to vote a particular way).)

4. Go learn about Karger's Algorithm and the divide-and-conquer improvement discussed above for minimum cut-set in an undirected graph.

5. Go learn about augmenting paths and pre-flow push for network flow.

6. Go learn about universal hashing and have confidence in your hash functions whenever you truly need it!