Code Jam 2016 - Round 3

Analysis: Forest University

Forest University: Analysis

An unusual problem

Small-only problems are relatively rare in Code Jam. Sometimes, as with 2012's Proper Shuffle from Round 1A 2014, this is because we have a randomized solution in mind and want contestants to be able to try again if necessary, even if the probability of a contestant's optimal solution actually failing due to chance is very small. The Forest University problem is Small-only for both of these reasons. We could have added another Small dataset small enough to solve precisely via dynamic programming, but we didn't think it would add useful resolution to help us pick our 26 advancers.

Clearly, with up to 100 courses, there are far too many course sequences to enumerate. The problem has an unusually relaxed tolerance for less precise answers, so it is a natural candidate for a simulation-based approach: we can generate course schedules and check whether they contain the cool substrings. Unusually for a simulation problem, though, the challenge here is to figure out how to do the simulation correctly even once! How do we sample the set of all course sequences uniformly at random?

A weighted-sampling method

Some pen and paper analysis of a small forest can reveal a simple rule that is sufficient to guide our simulations. Let's consider a case with five courses A, B, C, D, and E; A and E are basic, A is the prerequisite of both B and D, and B is the prerequisite of C.

Then there are fifteen possible orders in which to take the courses: ABCDE, ABCED, ABDCE, ABDEC, ABECD, ABEDC, ADBCE, ADBEC, ADEBC, AEBCD, AEBDC, AEDBC, EABCD, EABDC, EADBC. Notice that four-fifths of these begin with A and one-fifth begin with E. A is a root node with a total of 4 descendants (including itself); E is a root node with a total of 1 descendant (including itself). This is a promising pattern, and it is not coincidental! It suggests a method for building up a course string while sampling uniformly: keep choosing one of the available courses, with probability proportional to the size of that course's subtree (itself plus all its descendants).

To apply this method to the example above: we start with an empty course string, and at first, we can only choose either A or E. A has 4 descendants including itself, and E has 1, so we choose A with probability 4/5 and E with probability 1/5. Suppose that we choose A. Now we have three choices for the next course: B, D, and E. B has 2 descendants including itself; D has 1; E has 1. So, we choose B with probability 2/4, D with probability 1/4, and E with probability 1/4. Suppose that we choose D. Now we have two choices: B and E; we choose these with probability 2/3 and 1/3, respectively. We continue in this way until we've built a full sequence.

Why does this work? Speaking more generally: let's add a single root node to be the parent of all nodes with no prerequisites. We can now recursively compute, for each subtree, a schedule of just the classes in that subtree. Suppose that the subtree has root node V and size S. First, we compute a schedule for each subtree whose root is one of the children of V. Then, we put V first in the new schedule we are creating, and we choose an assignment (uniformly at random) of the remaining S-1 positions to each of the subtrees, such that each subtree is assigned as many positions as it has nodes. Then, we copy the ordering for each subtree into that subtree's positions. This results in a uniformly randomly chosen schedule.

Since we are picking a uniformly random assignment for each subtree, then interleaving them together uniformly randomly, the fraction of possible orders in which a given top-level course (i.e. the root of a certain subtree) appears first can be computed using a multinomial coefficient. For example, if we have to interleave A elements from one subtree and B elements from another subtree, the total number of ways to do so is (A+B)! / (A! × B!). The number of these ways that start with an element from subtree A is (A+B-1)! / ((A-1)! × B!), and the number of these ways that start with an element from subtree B is (A+B-1)! / (A! × (B-1)!). The ratio between these is (A! × (B-1)!) / ((A-1)! × B!), and this reduces to A / B. This explains why it suffices to sample proportionately to the size of each subtree.

A surprisingly elegant method

Generating a random sequence is equivalent to assigning distinct numbers from 1 to **N** to nodes in our forest, in such a way that the number of a node is smaller than the number of all its descendants. Let's start with any of the **N**! possible assignments, chosen uniformly. Now let's go through nodes from top to bottom, and if a node is not the smallest in its subtree, we swap its number with the smallest. This generates sequences uniformly, since each sequence can be obtained from P different permutations, where P is the product of sizes of subtrees of all nodes. The probability that a given node X is the smallest (once it's available to be chosen) is proportional to the size of the tree it is rooting, because any of those nodes that get the smallest number will "give" it to X.

But will we get a precise enough answer?

If we check *K* uniformly generated sequences, and the true probability to find a given substring is *p*, then the fraction of those *K* sequences that contain this substring follows a binomial distribution with parameters *K*, *p*, which we can approximate with a normal distribution with mean p and standard deviation sqrt(*p* × (1 - *p*) / *K*). (This Wikipedia article has some guidelines on when this approximation is valid.) If we run 10000 iterations, this is at most 0.5 / 100=5e-3, so the required precision of 3e-2 is 6 standard deviations. So, the probability of having one particular answer incorrect is roughly 1 in 500 million, which means that the probability of having at least one of the 500 required answers incorrect is at most 1 in a million.

The error bounds are generous enough that, with a fast enough implementation, the problem is solvable in slower languages such as Python.