Code Jam 2012 - Round 3

Analysis: Lost Password

This is certainly an intimidating problem. Even small strings, like the example from The Fellowship of the Ring, are difficult to work through. When you are trying to combine passwords of length at most 500 into a password string, there are so many possibilities to consider that optimization quickly becomes overwhelming.

The Small Input

Let's begin with the small input. In this case, we need to connect up pairs of letters. The key insight here is to imagine everything as a graph. If you have heard of De Bruijn sequences before, that is the perfect thing to build from. We will make a vertex for each letter (the 26 normal letters as well as all the 133t variations), and add an edge between every pair of letters. Each 2-letter word that we need in our password string corresponds to an edge on this graph. For example, "t0" corresponds to the edge 't' -> '0'. Let's call these *candidate edges*.

Next let's consider a password string. We can think of this as a path on the graph. We start at the first letter, and then continually move to the next letter in the string. For example, "abc0" corresponds to the path 'a' -> 'b' -> 'c' -> '0'. Therefore, the problem can be re-stated as follows: what is the length of the shortest path on this graph that includes all the candidate edges?

Here is it is helpful to think back to another classic algorithm problem: Eulerian paths. The problem could also be re-stated as: if we start with just the candidate edges, what is the minimum number of edges we need to add so that the resulting graph has an Eulerian path? Fortunately for us, the Eulerian path problem has been completely solved!

Fact: A directed graph has an Eulerian path if and only if (1) every vertex has in-degree equal to its out-degree except possibly for two vertices that are off by one, and (2) every vertex with positive degree is connected in the underlying undirected graph.

If you play with some examples, you will see that you will always be connected here, but let's come back to a formal proof when we talk about the large input. The fact that connectivity comes for free is really what makes this problem solvable, and so it is a good thing to think about!

The remaining condition says that all we need to do is add edges to balance in-degrees and out-degrees. We can add any edge we want at any time, so the method is simple: choose a vertex u with higher in-degree than out-degree and a vertex v with higher out-degree than in-degree, and then add an edge from u to v. Repeat until there are only two vertices left that are only slightly imbalanced, and we're done! After all that talk, we end up with a very simple greedy algorithm.

The Large Input

In fact, this solution already has most of the key ideas that go into solving the large as well, but we just need to take them further. There are three main challenges:

- When passwords are length greater than 2, how do we interpret them as edges on a graph?

- When it comes time to balancing degrees, some edges will be impossible to add. How do we adapt the greedy algorithm from before?
- The output password could be huge! Is it really possible to solve the problem in time less than the size of the output password?

The first challenge is the most important, and again, De Bruijn sequences provide a good model to build from. If we are trying to construct all passwords of length k, we will create a graph with vertices corresponding to all lengh-(k-1) phrases. Each candidate password corresponds to the edge between its prefix phrase and its suffix phrase. Let's begin by making this nice and rigorous, although the intuition is exactly the same as in the small input.

A Minimum-Cost Flow Interpretation

Let us call any length-k substring of **S** (and any of its "l33tspeak" equivalents) a "candidate", and any length-(k-1) string composed of digits and lowercase letters a "phrase" (whether it is contained in a candidate or not). For each phrase s, define its weight w(s) to be the number of candidates that end with s minus the number of candidates that begin with s. Note that the sum of w(s) is 0. (The weight of a phrase will measure how far the corresponding vertex is from supporting an Eulerian path.)

Form a directed graph G with vertices corresponding to the phrases. For phrases *a* and *b*, add an edge from *a* to *b* if you can make *a* into *b* by adding a letter to the end of *a* and removing the first letter of *a*. Now we set up a flow problem: if a phrase *s* has positive weight, it is a source with capacity |w(s)|; otherwise it is a sink with capacity |w(s)|. All edges have infinite capacity and cost one.

Let ANSWER denote the shortest possible length of a password string for **S**, let C denote the set of all candidates, and let FLOW_EDGES denote the minimum number of edges (i.e. minimum cost) in an "almost-maximum" flow. (Specifically, this is a flow that leaves at most 1 capacity unused in some source and at most one 1 capacity unused in some sink).

Lemma 1: ANSWER = FLOW_EDGES + |C| + k - 1.

Proof Part 1: ANSWER ≤ FLOW_EDGES + |C| + k - 1

Let G' be the directed graph formed as follows:

- Begin with the directed multi-graph formed by the minimum-cost almost-maximum flow on G.
- For each candidate c, add an edge from the prefix phrase of c to the suffix phrase of c.

For a given phrase s, let's calculate its out-degree minus its in-degree in G'. After the first step, this value is exactly w(s), except for two vertices that are off by 1. (This is due to the fact that our flow is only almost maximum.) After the second step, this value becomes exactly 0, again except for two vertices that are off by 1.

Therefore, we know G' satisfies the condition on in-degrees and out-degrees for it to have an Eulerian path. (See the "Fact" in the Small Input discussion.) We now show it also satisfies the connectivity condition. This is where we use the specific nature of the problem and the fact that |**S**| ≥ 2k. Actually, we suspect the last condition is unnecessary in the end, but the proof becomes much harder if this is not true!

Let's say a vertex s is a *core* vertex if it corresponds to a phrase in **S**, or to a 133tspeak variation of such a phrase.

- If s a core vertex, then s is adjacent in G' to its predecessor and successor phrases within **S**. (Note that there may be multiple predecessors or successors if the new letter has a leet variation or if **S** appears multiple times.) Therefore, we can certainly follow the edges of G' to go from s to some phrase a(s) that starts at the first letter of **S**. We can then walk from there to a phrase b(s) that ends at the last letter of **S**. Since a(s) and b(s) are completely disjoint, we can choose b(s) to be completely non-leet by always adding on non-leet successor letters. This means b(s) does not depend on s, and hence we have demonstrated every core vertex is connected to a single vertex in G'.
- Now consider a non-core vertex t with positive degree in G'. This can only happen if t has positive degree in G, and therefore it must be connected via the flow to some core vertex s. Since we just showed all core vertices are connected, we conclude the non-core vertices are connected as well.

Therefore, the underlying undirected graph of G' is indeed connected, and hence G' has an Eulerian path consisting of some vertices s_1, s_2, ... $s_{\text{FLOW\_EDGES + |C| + 1}}$. We can now construct a password string as follows:

- Begin the password string with the k - 1 letters in s_1.
- For each subsequent vertex s_i, append to the password string the one letter at the end of s_i that is not in s_{i-1}.

This string has length exactly k - 1 + FLOW_EDGES + |C|, as required. Notice that after appending the letter for s_i, the final k-1 letters in the password string are always precisely equal to s_i. (This invariant can easily be proven by induction.) Now consider an arbitrary candidate c. Because of how we constructed G', c has prefix s_{i-1} and suffix s_i for some i. But then after appending the letter for s_i, the last k letters precisely spell out c. Hence, every candidate is in this password string, and the inequality is proven.

Proof Part 2: ANSWER ≥ FLOW_EDGES + |C| + k - 1

For the second part of the proof, we just need to reverse the previous argument. It is actually a little easier because we do not have to worry about connectivity.

Consider a password string P of length ANSWER. By definition, we know each candidate must appear somewhere in P. Therefore, for each candidate c, we can define pos(c) to the be the character in P where the first occurrence of c begins. Now let's order the candidates c_1, c_2, ..., $c_{|C|}$ by pos(c).

For each i, consider the substring of P starting with character pos(c_i) + 1 and ending with character pos(c_{i+1}) + k-2. Note that this substring begins with the suffix phrase of c_i and ends with the prefix phrase of c_{i+1}.

By reversing the construction from the previous section, we can interpret this substring as a path in the graph G from the suffix phrase of c_i to the prefix phrase of c_{i+1}. This path has pos(c_{i+1}) - pos(c_i) - 1 edges in it. Now let's think about what happens when we combine all these paths. We get a flow with a source at every suffix phrase except $c_{|C|}$ and a sink at every prefix phrase except c_1. When we add all these sources and sinks together, we get precisely an almost max-flow on G. Furthermore, this flow uses exactly sum_i [pos(c_{i+1}) - pos(c_i) - 1] = pos($c_{|C|}$) - pos(c_1) - |C| + 1 edges.

It follows that FLOW_EDGES ≤ pos($c_{|C|}$) - pos(c_1) - |C| + 1. Finally, we know pos($c_{|C|}$) ≤ |P| - k = ANSWER - k, and also pos(c_1) ≥ 0. Therefore, ANSWER ≥ FLOW_EDGES + k + |C| - 1, as required.

A Greedy Solution

At this point, we still have not solved the whole problem, but we have reduced it to something more tractable. Minimum-cost flow is a complicated problem but it is well known, and at least it can be solved in polynomial time. We can either try to optimize here, or we can use more clever idea to make our life much simpler.

Lemma 2: Fix the graph G and assign arbitrary source and sink capacities to arbitrary nodes. Then, a minimum-cost almost-max flow can be achieved by repeatedly taking the shortest path from an unused source to an unused sink, and pushing flow along there (without ever reverting any flow that's already been pushed).

Proof: Let F denote a minimum-cost almost-max flow on G, and suppose the shortest path in G between a source and a sink goes from source u to sink x. Furthermore, suppose that F contains a path from u to a different sink y, and a path from a different source v to x. We claim that we could replace these two paths with a path from u to x and with a path from v to y to achieve a flow with no more edges. (Since F is only an almost-max flow, it might also be that u and/or x is simply not used in F, but that case is a lot simpler.) Recall that every edge in G has infinite capacity, so the only challenge here is making sure the path lengths work out.

Given two phrases p and q, let's define A(p, q) to be the longest string that is both a suffix of p and a prefix of q. Then the distance from p to q in G is precisely equal to k - 1 - |A(p, q)|. This means we can reformulate the claim as follows: given that |A(u, x)| ≥ max(|A(u, y)|, |A(v, x)|), prove that |A(u, x)| + |A(v, y)| ≥ |A(u, y)| + |A(v, x)|.

Now, notice that A(u, x) and A(u, y) are both suffixes of u, but A(u, x) is at least as long. Therefore, A(u, y) is a suffix of A(u, x). Similarly, A(v, x) is a prefix of A(u, x). Let t = |A(u, y)| + |A(v, x)| - |A(u, x)|. If t ≤ 0, the claim is trivial. Otherwise, there must be a length-t string z that is a suffix of A(v, x) and a prefix of A(u, y). Then z is also a suffix of v and a prefix of y. Therefore, |A(v, y)| ≥ |z| = t, and the claim is proven.

We have shown that F can be modified to contain the path from u to x without increasing the number of edges. Now consider the rest of F. It defines a smaller flow, and we can repeat the same argument to show this residual flow can also be modified to contain the shortest path between a remaining source and a remaining sink, and that this modification will not increase the number of edges. Applying this argument repeatedly gives us the flow proposed in the lemma without ever increasing the number of edges, and so we have shown that this flow is indeed optimal.

The Implementation

Almost there! We have outlined a greedy algorithm, but what does it actually mean when it is put back into the context of the original problem, and how can it be implemented quickly?

- The first thing to do is to construct the prefix phrases and the suffix phrases of all candidates. We need to construct an almost max-flow from the suffixes to the prefixes. If a phrase is both a suffix and a prefix, then these two instances cancel out.
- First we should look for a source u and a sink x that are separated by distance 1 in the underlying graph. This is equivalent to saying that there is a length k-2 string that is a suffix of u and a prefix of x.
- Next we should look for a source u and a sink x that are separated by distance 2 in the underlying graph. This is equivalent to saying that there is a length k-3 string is a suffix of u and a prefix of x.
- And so on...

This can be summarized as follows:

- Let P = the multi-set of prefix phrases of all candidates.
- Let S = the multi-set of suffix phrases of all candidates.
- Let x = k + |C| and i = 0.
- (*) While |P| ≥ 2 and |P intersect S| ≥ 1: delete one copy of a common element from both P and S and increment x by i.
- Remove the last letter from every element in P and the first letter from every element in S.
- Increment i by 1, and repeat again from (*) until P and S have just one element.
- Output x.

Unfortunately, this is still a little too slow. It will run in time proportional to the output password string length, which could be 10^{18}. The final optimization is that if you look at any element in P (or S) in the algorithm above, then all 133tspeak variations of that element should also be in P (or S). You can always treat all variations as one big batch:

- Let P = the map from phrase to the number of times a 133tspeak variation of that phrase appears as the prefix of a candidate.
- Define S similarly for suffixes.
- Let x = k + |C| and i = 0.
- (*) While P and S are non-empty:
- While P and S have a common element t: delete min(P[t], S[t]) from P[t] and S[t] and increment x by i * min(P[t], S[t]).
- Remove the last letter from every element in P and the first letter from every element in S.
- Increment i by 1 and repeat again from (*) until P and S are empty.
- Output x-i+1.

The Misof Implementation

Only one contestant solved this problem during the contest, and he used a variation on the method approached here. Instead of using the greedy algorithm, he implemented the flow directly, grouping together 133t variations in the same way that we did to achieve sub-linear running time in the password string size. It is a little slower and a little harder to implement, but it works just fine. Congratulations to misof for coming up with this!