Code Jam 2020 - Virtual World Finals

Analysis: Replace All

The first thing we can notice is that since we replace every occurrence of a character, and at the end we count unique characters, we can regard the input **S** as a set. That is, multiple occurrences of the same character can be ignored, and the order of the characters is unimportant. We write "x → y" to represent the replacement of a character x by a character y. Here, and in the rest of the analysis, we use both lowercase and uppercase letters to represent variables, not actual letters that can happen in the input.

To simplify the explanations, we can represent the list of replacements as a directed graph G where nodes represent characters, and there is an edge from x to y if and only if there exists a replacement x → y in the input. Notice that each weakly connected component of G defines a problem that can be solved independently, and combining the results corresponds to adding up the number of distinct characters that can occur in the final text from each group. In particular, notice that a character c that does not appear in any replacement forms a single-node connected component, and the result for each such character is either 1 if c is in **S**, and 0 otherwise.

Let us define *diversity* as the number of unique characters in a text. Diversity of the last text is exactly what we are trying to maximize. If both x and y are in U, the diversity after performing x → y on U is 1 less than before. On the other hand, if at least one of x and y is not in U, then the diversity before and after performing x → y is the same. That is, the replacement can either cause a diversity loss of 1, or not cause any loss. We can then work on minimizing the diversity loss, and the final answer will be the original diversity minus the loss.

Test Set 1

In Test Set 1 the in-degree of each node of G is limited to at most 1. This restricts the types of weakly connected components the graph can have. If there is a node in the component with in-degree equal to 0, then the component is a directed tree and the node with in-degree equal to 0 is the root. If all nodes in the component have in-degree equal to 1, then there is a cycle (which we could find by starting at any node and following edges backwards until we see a node for the second time). Each node in the cycle only has its incoming edge coming from another node in the cycle, but it can have edges pointing out to non-cycle nodes. This means each cycle-node can be the root of a directed tree. These are the connected components of pseudoforests.

We can break down this test set by component type. Let's consider the simplest possible tree first: a simple path $c_1 \rightarrow c_2 \rightarrow ... \rightarrow c_k$. If both c_{k-1} and c_k are in the initial text **S**, then we cannot perform $c_{k-1} \rightarrow c_k$ without losing diversity. No other replacement in the text can make c_{k-1} or c_k disappear from the text, so they will both be present until $c_{k-1} \rightarrow c_k$ is performed for the first time. In addition, after we perform $c_{i-1} \rightarrow c_i$ for any i, we can immediately perform $c_{i-2} \rightarrow c_{i-1}$ without loss of diversity, since c_{i-1} will definitely not be in the text then. Therefore, performing every replacement in the path in reverse order works optimally.

With some care, we can extend this to any tree. Consider a leaf node y that is maximally far from the root. The replacement x → y that goes into it is in a similar position as the last replacement in a path, but there could be other replacements that remove x from the text before performing x → y, which would prevent x → y from causing diversity loss. Since y is maximally far from the root, however, all other replacements that can help also lead to leaf nodes. If x and the right side of all replacements that go out of x are in **S**, there is no way to avoid losing diversity. However, only the first one we perform will cause diversity loss, whereas all the others can be done when x is no longer in the text, preventing further loss. We can generalize this by noticing that we can "push" characters down the tree without loss of diversity as long as there is at least one descendant node that is not in the current text. This is done by considering the path from a node x to the descendant that is not in the text y, and processing the path between x and y as described earlier.

From the previous paragraph, we can see that any edge x → y that does not go into a leaf node can be used without diversity loss. Even if there are no descendants of x in the tree that are not in the initial text **S**, we can process some unsalvageable leaf node first, and then we will have room to save x → y. Any edge z → w that goes into leaf nodes may be salvageable as well, as long as z has other descendants. Putting it all together, we can process the nodes in reverse topological order. At the time when we process a node x, if there is any descendant of x that is not in the current text, then we can push x down and then process all of x's outgoing edges without any diversity loss. If we process a node x and its subtree consists only of nodes representing letters currently in the text, that means that we will lose 1 diversity with the first edge going out of x that we process (it doesn't matter which one we process first), but that loss is unpreventable.

To handle cycles, consider first a component that is just a simple cycle. If there is at least one node representing a character x not in **S**, then we can process it similarly to paths, without diversity loss: we replace y → x, then z → y, etc, effectively shifting which characters appear on the text, but keeping diversity constant. If all characters in the cycle are in **S**, then the first replacement we perform will definitely lose diversity, and after that we can prevent diversity loss by using the character that disappeared as x in the process from the first case.

If a component is a cycle with one or more trees hanging from it, we can first process each tree independently. Then, each of those trees will have at least one node representing a character not in the current text: either there was such a node from the start, or it was created as part of processing the tree. So, we can push down the root of one of those trees (which is a node in the cycle) without losing diversity. Then, there will be at least one node in the cycle whose represented character is not in the text, so we can process it without diversity loss.

Test Set 2

In Test Set 2, the graph G can be anything, so we cannot do a component type case analysis like we did for Test Set 1. However, we can still make use of the idea of processing paths and pushing characters to avoid diversity loss following our first replacement.

We tackle this by morphing the problem into equivalent problems. We start with the problem of finding an order of the edges of G that uses each edge at least once while minimizing steps in which we cause diversity loss.

First, notice that once we perform x → y, we can immediately perform any other x → z without diversity loss. So, if we have a procedure that uses at least one replacement of the form x → y for each x for which there is at least one, then we can insert the remaining replacements without altering the result. That is, instead of considering having to use every edge at least once, we can simply find a way to use every node that has out-degree at least 1. Furthermore, any replacement x → y such that x is not in **S** can be performed at the beginning without changing anything, so we do not need to worry about using them. Notice that we are only removing the requirement of using particular edges, not the possibility. Therefore, we can make the requirements even less strict: now we only need to use every node with out-degree at least 1 that represents a character present in **S**.

Now that we have a problem in which we want to cover nodes, and we know from our work in Test Set 1 that we can process simple subgraphs like paths and cycles in a way that limits diversity loss, we can define a generalized version of the minimum path cover problem that is equivalent.

Given a list of paths L, let us define the *weight* of L as the number of paths in L minus the number of characters c not in **S** such that at least one path in L ends with c. That is, for each character c not in **S**, we can include just one path ending in c in L without adding to its weight. Let us say that a list of paths L *covers* G if every node in G with out-degree at least 1 that represents a character in **S** is present in some path in L in a place other than at the end. That is, at least one edge going out of the node was used in a path in L. We claim that the minimum weight of a cover of G is equal to the minimum number of steps that cause diversity loss in the relaxed problem.

To prove the equivalency, we first prove that if we have a valid solution to the problem that causes D diversity loss, we can find a cover that has weight at most D. Afterwards we prove that if we have a cover with weight D, we can find a solution to the problem that causes a diversity loss of at most D.

Let r_1, r_2, ..., r_n be an ordered list of replacements that satisfies the relaxed conditions of the problems and has D steps that cause diversity loss. We build a list of paths L iteratively, starting with the empty list. When considering replacement $r_i = x \rightarrow y$:

1. If the text before and after r_i are the same, we do not change L.
2. If there is a path starting with y in L, we append x at the start of it.
3. Otherwise, we add r_i as a new path in L.

Notice that the first time we perform a replacement $x \rightarrow y$ where x is a node that we need to cover, we cannot be in case 1: the xs that were originally in **S** have not been replaced yet, so r_i will replace them and change the text. Cases 2 and 3 add x as a non-final part of a path. Further processing can only append things to the start of the path, so x will remain a non-final member of it. This proves that L is a cover.

Now, suppose step $r_i = x \rightarrow y$ falls into case 3 and adds a path to L that increases its weight. Because we are not in case 1, x is in the text at the time we begin to perform r_i. Since we are in case 3, there wasn't a path in L starting with y. That means that for any previous replacement of the form $y \rightarrow z$ that erased ys from the text, there was another replacement of the form $w \rightarrow y$ that that reintroduced y and caused it to no longer be a starting element of the path in L in which $y \rightarrow z$ was added. Moreover, since we are assuming this step increases L's weight, either y was in **S** or there is a path in L ending with $v \rightarrow y$, which introduced y into the text. In either case, y is in the text before performing r_i, meaning that r_i causes diversity loss. Since every step that adds weight to L is a step that causes diversity loss, the weight of L is at most D.

Now let us prove that given a cover L of weight D, we can find a solution to the problem with diversity loss D. For that, we need to process paths in a manner that is not as simple as the one we used for Test Set 1. The way in which we process paths in Test Set 1 could only result in diversity loss on the first replacement made, but it also produced other changes in the text. Those changes could hurt us now that we have to process multiple paths within the same connected component.

To process a path P, we consider the status of the text U right before we start. We call a node *strictly internal* to a path if it is a node in the path that is neither the first nor the last node in the path. We split P into ordered subpaths P_1, P_2, ..., P_n such that the last node of P_i is the same as the first node of P_{i+1} (the edges in the subpaths are a partition of the edges in P). We split in such a way that a strictly internal node of P is also a strictly internal node of the P_i where it lands if and only if the character it represents is in U. Then, we process the subpaths in reverse order P_n, then P_{n-1}, ..., then P_1. Within a subpath, we perform the replacements in the path's order (unlike in Test Set 1). Since the intermediate characters within a subpath do not appear in S, performing all the replacements of a subpath that starts with x and ends with y has the net effect of replacing x by y. In the case of P_1, if x is not in U, then there is no effect. After processing subpath P_{i+1} that starts with x_{i+1}, x_{i+1} is not in U, and after processing subpath P_i, x_i is not in U and x_{i+1} is restored. The net effect is that processing the path in this way effectively replaces the first character of the path that is in U by the last character of the path, and doesn't change any other character.

Let L' be a sublist of L consisting of exactly one path that ends in c for each character c not in **S**. L - L' contains exactly D paths. We process the paths in L' first, and then the ones in L - L'. When we process a path in L', we change the text by introducing a new character to it. However, that character is not in the original **S**, and it is always a new one, so those changes cause no diversity loss. When processing all other paths, since the net effect is that of a single replacement, we can cause at most 1 diversity loss each time, which means at most D diversity loss overall.

Notice that if we have a path that touches a node x that represents a character in **S**, we can replace that node in the path with a cycle that starts and ends at x and visits its entire strongly connected component. A replacement like this one on a path in a list does not change its weight. Therefore, we can relax the condition to require covering just one node of out-degree at least 1 and with a represented character in **S** per strongly connected component.

We can turn this into a maximum matching problem on a bipartite graph similarly to how we solve the minimum path cover problem in directed acyclic graphs. Consider a matching M between the set of strongly connected components C (restricted to nodes representing characters in **S**) that need to be covered, and the set D equal to C and all remaining nodes representing characters not in **S**. An element c from C can be matched to an element d from D if there is a non-empty path from c to d (it cannot be matched with itself). This matching represents a "next" relationship assignment, and unmatched elements represent "ending" nodes in paths. More formally, let f(c) be the function that maps a member of C to the member of D that corresponds to the same strongly connected component. We can create a cover with weight exactly the size of the unmatched elements from C by creating a path that adds weight for each unmatched element from C, and adding paths that do not add weight for each element in D - C.

For each unmatched element c from C, add it to a new path, then append to its left its matched element M(f(c)), then M(f(M(f(c)))), etc. Then, for each matched element c in D - C, which are the characters not in **S**, do the same. This yields a set of paths that touches every element of C, and the ending elements are either unmatched elements from C or characters not in **S** which do not add weight. Notice that some unmatched elements could yield single node paths, which are paths not really considered above. However, since such a path always counts toward the weight, we can append any replacement to its end to make it not empty without increasing the weight.

Therefore, the size of the unmatched elements from C in a maximum matching of the defined relation, which we can calculate efficiently by adapting a maximum flow algorithm like Ford-Fulkerson, is equal to the minimum weight of a cover, which is what we needed to calculate.

Despite the long proofs, the algorithm is relatively straightforward to implement. The graph can be built in time linear in the size of the input, while the strongly connected components and their transitive closure, which are required to build the relation for the matching, can be found in quadratic time in the size of the alphabet A (the relation itself can have size quadratic in A, so it cannot be done faster). The maximum matching itself takes time cubic in A, because it needs to find up to A augmenting paths, each of which can take up to $O(A^2)$ time (linear in the size of the relation graph). This leads to an overall complexity of $O(A^3)$, which is fast enough for the small alphabet size of this problem. Moreover, we could use something simpler and technically slower for the strongly connected components and their transitive closure like Floyd-Warshall, simplifying the algorithm without affecting the overall running time.