1 UVA Problem 124: Following Orders

Background

Order is an important concept in mathematics and in computer science. For example, Zorn's Lemma states: "a partially ordered set in which every chain has an upper bound contains a maximal element." Order is also important in reasoning about the x-point semantics of programs. This problem involves neither Zorn's Lemma nor x-point semantics, but does involve order.

The Problem

Given a list of variable constraints of the form x <y , you are to write a program that prints all orderings of the variables that are consistent with the constraints. For example, given the constraints x <y and x <z there are two orderings of the variables x, y, and z that are consistent with these constraints: x y z and x z y .

The Input

The input consists of a sequence of constraint specifications. A specification consists of two lines: a list of variables on one line followed by a list of contraints on the next line. A constraint is given by a pair of variables, where x y indicates that x < y. All variables are single character, lower-case letters. There will be at least two variables, and no more than 20 variables in a speci cation. There will be at least one constraint, and no more than 50 constraints in a speci cation. There will be at least one, and no more than 300 orderings consistent with the contraints in a speci cation. Input is terminated by end-of- le.

The Output

For each constraint speci cation, all orderings consistent with the constraints should be printed. Order- ings are printed in lexicographical (alphabetical) order, one per line. Characters on a line are separated by whitespace.

Output for different constraint specifications is separated by a blank line

1.1 Mathematical Formulation

Given an input of N distinct letters ($2 \leq \mathrm{N} \leq 26$) and a set of R rules ($1 \leq \mathrm{R} \leq 50$). The algorithm will enumerate all M anagrams of the letters adhearing to the rules. $Note:\ worst\ case\ M \simeq N!$

1.2 Solution

Important Confusing Data Structures:

- LinkedList<Integer>[n] **G** : Each array slot corresponds to the symbol that appears in that index in the sorted version of the input. Each slot holds a linked list of the symbols that must appear after the given symbol according to the user inputted rules.
- int[n] **inDegree** : This is an array that stores (at each recursive step) the number of symbols that must come before it. If the index is -1 $\implies$ that this was seen already in the recursion and has already been recorded within this branch of recursion.
- char[n] **result** : This is an array storing the symbols being used so far in the recursive step. It is being used throughout all of the different branches and never copied.

In the first algorithm, the program initializes all of the data structures that will be used, filling in the pertainant information. Once the structures have been properly formed, the program will then call it's recursive method which will enumerate the valid strings in alphabetic order. If there is another case, it will reinitialize everything appropriately and repeat, otherwise it will terminate.

Algorithm 1 FollowingOrders main

```
procedure MAIN( )
    map ← new int[26] // to store the index of each letter
    while inFile.HASNEXT( ) do // each case
        symbols ← sorted char[n] of the symbols
        comparisons ← char[2*r] of all the rules, each pair is a rule
        inverseMap ← char[n]
        for (i ∈ symbols.length do
            Fill in map and inverseMap using symbols
        G ← LinkedList<Integer>[n] all of the rules for each letter
        inDegree ← int[n] // stores indegree for each symbol
        // Fill in G and inDegree
        for i ∈ comparisons.length do
            less ← map[comparisons[i]]
            greater ← map[comparisons[i+1]]
            G[less].ADD(greater)
            inDegree[greater]++
        results ← char[n] // to store the results thusfar in recursion
        DFSHASH(n, inverseMap, inDegree, G, result)) // See Below
```

The main functionality of dfsHash is to do a depth first search recursively on the valid strings in alphabetic order. It accomplishes this through using the combination of the data structures explained above, inDegree and G. The way that they work is that, the algorithm will look through inDegree for the first occurance of a $0 \implies$ the first letter that can go first (unimpeeded by rules) and record it in the first index of result. Then the algorithm will look at all of the letters that must come after it (found in G). For each of these letters, their inDegree will be decrimented by 1 soas to suggest that one of the letters coming before it has been written down. Then a recursive call is made, attempting to find the second letter in the same manner...etc. Once the first word has been found (all given symbols have been used up) the algorithm will back track and (from the second to last letter) look for the next occuring 0 in the array. This will be the second word. If there is none, it will kill this branch and return back to the previous step. *Note* : *The recursive call will re − increment all decrimented indicies within inDegree*

Algorithm 2 FollowingOrders Recursive Method

```
procedure dfsHash(left,inverseMap[],inDegree[],G,result[])
    // base case, this means we have all n letters in result
    if left == 0 then
        print(result)
    for i ∈ results.length do
        if inDegree[i] == 0 then
            inDegree[i] ← -1
            results[results.length − left] ← inverseMap[i]
            for letter l ∈ G[i] do
                inDegree[letter]-
            dfsHash(left-1, im, inDegree, G, result)
            for letter l ∈ G[i] do
                inDegree[letter]++
            inDegree[i] ← -1
```

1.3 Correctness

Proposition 1.

One call of the dfsHash method will be able to explore all possible anagrams from this position in alphabetical order while adhearing to the rules.

Proof.

Using the fact that the inDegree array corresponds to how many non-used letters are valid at this stage; we will exhibit this by saying that an entry can be denoted as $d_0, d_1, ...d_k$ for $(k+1)$ distinct letters and d_i<0 $\implies$ *already used*, $d_i = 0$ $\implies$ *valid letter and* $d_i = j$>0 $\implies$ j *non − used letters must come before it*. Since inDegree is the representation of all of the letters in alphabetic order $\implies$ the first 0, is the first valid letter, the next 0 is the next valid letter and so on and so forth. $\therefore$ we see that we will do all of the first possible ones first and then (starting from the last letter) we will work our way back up continuing our depth first search. This will do an in-order traversal of all the valid anagrams that can be produced. □

1.4 Analysis

For the following analysis, we will say that Given an input of N distinct letters (2 ≤ N ≤ 26) and a set of R rules (1 ≤ R ≤ 50). The algorithm will enumerate all M anagrams of the letters adhearing to the rules. *Note* : *worst case* $M \simeq N!$

Proposition 2. *The space complexity of this algorithm is* ***O(N + R)***

Proof.

This is due to the fact that all of our data is stored in 7 data structures:

- map: int array of size 26 (worst case for N)
- inverseMap: char array of size N
- symbols: char array of size N
- comparisons: char array of size $2 \cdot R$
- G: Integer Linked List Array which has size N and R Nodes
- inDegree: int array of size N
- results: char array of size N

The only points of contension here have to do with our recursive calls on inverseMap, G, inDegree, and results. If a new copy were made each call, we would have at most N of each of these, however since we are passing the object as a parameter, we are only using the same pointer, pointing to a single object for each so the same one is being accessed and modified each time. This allows for the algorithm to use only one object for each. Therefore, summing all of these up, we have a space complexity of $\mathrm{S}(N + N + N + 2 \cdot R + N + R + N + N) = \mathrm{S}(6 \cdot N + 3 \cdot R)$

Giving us a space complexity of **O(N + R)**

□

Proposition 3. *The time complexity of this algorithm is* ***O(M)***

Proof. This is the case because our algorithm reads in all of the input in linear time O(N + R) which are both trivially small. Then, for every recursive call, we search through the list of symbols (N) once. So we look at the case now that we are enumerating a word which is $c_0c_1c_2...c_n$ (W.L.O.G.)lets assume the only rule is that $c_0<c_k$ which would be the largest amount of anagrams possible. We can say that anagrams starting with c_0 number roughly $(k-1)!$ and for each letter, we will only do k^2 k's. $\implies$ going through all of the letters we would only go through k k^3 times $\implies$ k^4, replacing k with N we have N^4 yeilding a complexity of $M + N^4 + N + R$

Giving us a time complexity of **O(M)**

□

1.5 An Example

I did not have enough time to finish this section. However I have the correct output for the given example by the UVA site.