Algorithm for minimum number of characters that need to be changed in String S

[0] [2] user927026

[2018-08-28 06:39:02]

[algorithm dynamic-programming]

[https://stackoverflow.com/questions/52051567/algorithm-for-minimum-number-of-characters-that-need-to-be-changed-in-string-s]

I came across a programming challenge few days back which is over now. The question said given a string S of lowercase English alphabets, find minimum count of characters that need to be changed in String S, so that it contains the given word W as a substring in S.

Also in next line , print the position of characters you need to change in ascending order. Since there can be multiple output, find the position in which first character to change is minimal.

I tried using LCS but could only get count of characters that need to be changed. How to find the position of the character? I might be missing something, please help. Might be some other algorithm to solve it.

You are allowed to remove or add characters, correct? - **Olivier Melançon**
I can't remove characters. I can only replace the existing character. For eg if string S= "itisworstcap" ,and word W= "worldcup", then count =3 and position of char to change is 8 9 11 - **user927026**
In your example, you added characters, no? - **Olivier Melançon**
@OlivierMelançon To change `worstcap` to `worldcup`, you change `s` to `l`, `t` to `d` and `a` to `u`. No letters are added or removed. - **user3386109**
Replace "s" ->"l", "t"->"d","a"->"u". These replacements make the string S = "itisworldcup" - **user927026**
@user3386109 so the word 'itisworstcap' is simply an error, you meant 'worstcap'? - **Olivier Melançon**
@dreamhigh anyhow, if you are not allowed to swap, add or remove characters, you simply need to find the indices where the characters are not the same in both strings. - **Olivier Melançon**
@OlivierMelançon No, the string `S` ("itisworstcap") needs to be modified so that the word `W` ("worldcup") is a substring of `S`. - **user3386109**
@user3386109 my bad, I missed the part where you mentionned substrings, let me write down an answer - **Olivier Melançon**
@dreamhigh Did the challenge specify the time complexity of the algorithm. Given that `N` is the length of `S`, and `M` is the length of `W`, the obvious algorithm has complexity O(N*M). Because you can simply try the word at every possible starting position. - **user3386109**
The challenge is to solve it in least time. - **user927026**

[+2] [2018-08-28 08:01:09] Nico Schertler

The obvious solution is to shift the reference word `W` over the input string `S` and count the differences. However, this will become inefficient for very long strings. So, how can we improve this?

The idea is to target the search at places in `S` where it is very likely that we have a good match

with `W`. Finding these spots is the critical part. We cannot find them both efficiently and accurately without performing the naive algorithm. So, we use a heuristic `H` that gives us a lower bound on the number of changes that we have to perform. We calculate this lower bound for every position of `S`. Then, we start at the position of lowest `H` and check the actual difference in `S` and `W` at that position. If the next-higher `H` is higher than the current difference, we are already done. If it is not, we check the next position. The outline of the algorithm looks as follows:

```
input:
    W of length LW
    S of length LS

H := list of length LS - LW + 1 with tuples [index, mincost]
for i from 0 to LS - LW
    H(i) = [i, calculate Heuristic for S[i .. i + LW]]
order H by mincost
actualcost = infinity
nextEntryInH = 0
while actualcost >= H[nextEntryInH].minCost && nextEntryInH < length(H)
    calculate actual cost for S[H[nextEntryInH].index .. + LW]
    update actualcost if we found a lesser cost or equal cost with an earlier difference
    nextEntryInH++
```

Now, back to the heuristic. We need to find something that allows us to approximate the difference for a given position (and we need to guarantee that it is a lower bound), while at the same time being easy to calculate. Since our alphabet is limited, we can use a histogram of the letters to do this. So, let's assume the example from the comments: `W = worldcup` and the part of `S` that we are interested in is `worstcap`. The histograms for these two parts are (omitting letters that do not occur):

```
         a c d l o p r s t u w
worldcup 0 1 1 1 1 1 1 0 0 1 1
worstcap 1 1 0 0 1 1 1 1 1 0 1
------------------------------
abs diff 1 0 1 1 0 0 0 1 1 1 0  (sum = 6)
```

We can see that half of the sum of absolute differences is a proper lower bound for the number of letters that we need to change (because every letter change decreases the sum by 2). In this case, the bound is even tight because the sum is equal to the actual cost. However, our heuristic does not consider the order of letters. But in the end, this is what makes it efficiently calculatable.

Ok, our heuristic is the sum of absolute differences for the histograms. Now, how can we calculate this efficiently? Luckily, we can calculate both the histograms and the sum incrementally. We start at position 0 and calculate the full histograms and the sum of absolute differences (note that the histogram of `W` will never change throughout the rest of the runtime). With this information, we can already set `H(0)`.

To calculate the rest of `H`, we slide our window across `S`. When we slide our window by one letter to the right, we only need to update our histogram and sum slightly: There is exactly one new letter in our window (add to the histogram) and one letter leaves the window (remove from the histogram). For the two (or one) corresponding letters, calculate the resulting change for the sum of absolute differences and update it. Then, set `H` accordingly.

With this approach, we can calculate our heuristic in linear time for the entire string `S`. The

heuristic gives us an indication where we should look for matches. Once we have it, we proceed with the remaining algorithm as outlined at the beginning of this answer (start the accurate cost calculation at places with low heuristic and continue until the actual cost exceeds the next-higher heuristic value).

1

---

[0] [2018-08-28 07:42:46] arenard

LCS (= longest common subsequence) will not work because the common letters in W and S need to have matching positions. Since you are only allowed to update and not remove/insert.

If you were allowed to remove/insert, the Levenshtein distance could be used: https://en.wikipedia.org/wiki/Levenshtein_distance

In your case an obvious bruteforce solution is to match W with S at every position, with complexity O(N*M) (N size of S, M size of W)

2
