Edit Distance

Given two words *word1* and *word2*, find the minimum number of steps required to convert *word1* to *word2*. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

a) Insert a character
b) Delete a character
c) Replace a character

Solution 1

This is a classic problem of Dynamic Programming. We define the state `dp[i][j]` to be the minimum number of operations to convert `word1[0..i - 1]` to `word2[0..j - 1]`. The state equations have two cases: the boundary case and the general case. Note that in the above notations, both `i` and `j` take values starting from `1`.

For the boundary case, that is, to convert a string to an empty string, it is easy to see that the mininum number of operations to convert `word1[0..i - 1]` to `""` requires at least `i` operations (deletions). In fact, the boundary case is simply:

1. `dp[i][0] = i`;
2. `dp[0][j] = j`.

Now let's move on to the general case, that is, convert a non-empty `word1[0..i - 1]` to another non-empty `word2[0..j - 1]`. Well, let's try to break this problem down into smaller problems (sub-problems). Suppose we have already known how to convert `word1[0..i - 2]` to `word2[0..j - 2]`, which is `dp[i - 1][j - 1]`. Now let's consider `word[i - 1]` and `word2[j - 1]`. If they are euqal, then no more operation is needed and `dp[i][j] = dp[i - 1][j - 1]`. Well, what if they are not equal?

If they are not equal, we need to consider three cases:

1. Replace `word1[i - 1]` by `word2[j - 1]` (`dp[i][j] = dp[i - 1][j - 1] + 1 (for replacement)`);
2. Delete `word1[i - 1]` and `word1[0..i - 2] = word2[0..j - 1]` (`dp[i][j] = dp[i - 1][j] + 1 (for deletion)`);
3. Insert `word2[j - 1]` to `word1[0..i - 1]` and `word1[0..i - 1] + word2[j - 1] = word2[0..j - 1]` (`dp[i][j] = dp[i][j - 1] + 1 (for insertion)`).

Make sure you understand the subtle differences between the equations for deletion and insertion. For deletion, we are actually converting `word1[0..i - 2]` to `word2[0..j - 1]`, which costs `dp[i - 1][j]`, and then deleting the `word1[i - 1]`, which costs `1`. The case is similar for insertion.

Putting these together, we now have:

1. `dp[i][0] = i`;
2. `dp[0][j] = j`;
3. `dp[i][j] = dp[i - 1][j - 1]`, if `word1[i - 1] = word2[j - 1]`;
4. `dp[i][j] = min(dp[i - 1][j - 1] + 1, dp[i - 1][j] + 1, dp[i][j - 1] + 1)`, otherwise.

The above state equations can be turned into the following code directly.

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.length(), n = word2.length();
        vector<vector<int> > dp(m + 1, vector<int> (n + 1, 0));
        for (int i = 1; i <= m; i++)
            dp[i][0] = i;
        for (int j = 1; j <= n; j++)
            dp[0][j] = j;
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (word1[i - 1] == word2[j - 1])
                    dp[i][j] = dp[i - 1][j - 1];
                else dp[i][j] = min(dp[i - 1][j - 1] + 1, min(dp[i][j - 1] + 1,
dp[i - 1][j] + 1));
            }
        }
        return dp[m][n];
    }
};
```

Well, you may have noticed that each time when we update `dp[i][j]`, we only need `dp[i - 1][j - 1], dp[i][j - 1], dp[i - 1][j]`. In fact, we need not maintain the full `m*n` matrix. Instead, maintaing one column is enough. The code can be optimized to `O(m)` or `O(n)` space, depending on whether you maintain a row or a column of the original matrix.

The optimized code is as follows.

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.length(), n = word2.length();
        vector<int> cur(m + 1, 0);
        for (int i = 1; i <= m; i++)
            cur[i] = i;
        for (int j = 1; j <= n; j++) {
            int pre = cur[0];
            cur[0] = j;
            for (int i = 1; i <= m; i++) {
                int temp = cur[i];
                if (word1[i - 1] == word2[j - 1])
                    cur[i] = pre;
                else cur[i] = min(pre + 1, min(cur[i] + 1, cur[i - 1] + 1));
                pre = temp;
            }
        }
        return cur[m];
    }
};
```

Well, if you find the above code hard to understand, you may first try to write a two-column version that explicitly maintains two columns (the previous column and the current column) and then simplify the two-column version into the one-column version like the above code :-)

written by jianchao.li.fighter original link here

Solution 2

Use f[i][j] to represent the shortest edit distance between word1[0,i) and word2[0, j). Then compare the last character of word1[0,i) and word2[0,j), which are c and d respectively (c == word1[i-1], d == word2[j-1]):

if c == d, then : f[i][j] = f[i-1][j-1]

Otherwise we can use three operations to convert word1 to word2:

(a) if we replaced c with d: f[i][j] = f[i-1][j-1] + 1;

(b) if we added d after c: f[i][j] = f[i][j-1] + 1;

(c) if we deleted c: f[i][j] = f[i-1][j] + 1;

Note that f[i][j] only depends on f[i-1][j-1], f[i-1][j] and f[i][j-1], therefore we can reduce the space to O(n) by using only the (i-1)th array and previous updated element(f[i][j-1]).

```
int minDistance(string word1, string word2) {

        int l1 = word1.size();
        int l2 = word2.size();

        vector<int> f(l2+1, 0);
        for (int j = 1; j <= l2; ++j)
            f[j] = j;

        for (int i = 1; i <= l1; ++i)
        {
            int prev = i;
            for (int j = 1; j <= l2; ++j)
            {
                int cur;
                if (word1[i-1] == word2[j-1]) {
                    cur = f[j-1];
                } else {
                    cur = min(min(f[j-1], prev), f[j]) + 1;
                }

                f[j-1] = prev;
                prev = cur;
            }
            f[l2] = prev;
        }
        return f[l2];

    }
```

Actually at first glance I thought this question was similar to Word Ladder and I tried to solve it using BFS(pretty stupid huh?). But in fact, the main difference is that there's a strict restriction on the intermediate words in Word Ladder problem, while there's no restriction in this problem. If we added some restriction on intermediate words for this question, I don't think this DP solution would still work.

written by eaglesky1990 original link here

Solution 3

http://www.stanford.edu/class/cs124/lec/med.pdf

written by vikram.kuruguntla original link here