最小编辑距离

1 问题引入

最小编辑距离是从一个词变换到另一个词所需要的最少单字符操作总数（如插入，删除和替换）。其中，插入和删除的开销为1， 替换的开销为2。

例如，把单词`maximize`变为单词`minimum`的最小编辑距离为9，变换步骤如下：

```
Step 1 delete "a": "maximize" => "mximize"
Step 2 delete "x": "mximize" => "mimize"
Step 3 insert "n": "mimize" => "minmize"
Step 4 insert "i": "minmize" => "minimize"
Step 5 delete "i": "minimize" => "minimze"
Step 6 replace "z" with "u": "minimze" => "minimue"
Step 7 replace "e" with "m": "minimue" => "minimum"
```

2 计算模型

编辑距离问题拥有**最优子结构**，也就是说该问题能被分解成几个简单的子问题进行求解，子问题也能被分解为更简单的子问题，最后的子问题将很容易求解。下面给出编辑距离问题的描述及其子问题的描述，对不同的情形进行分析：

问题描述：通过对字符串`str1`进行编辑操作，把字符串`str1[1:m]`转换成字符串`str2[1:n]`。

子问题描述：通过对子串`str1`进行编辑操作，把子`str1[1:i]`转换成字符串`str2[1:j]`。

情形1：到达任一字串的末尾。

如果子串`str1`是空串，我们只需要把`str2`中剩余的字符都插入到`str1`当中即可，开销为`str2`中剩余字符的数量。如：

```
"", "ABC" => "ABC", "ABC"
cost = 3
```

如果`str2`是空串，结论同上。

情形2：子串`str1`和`str2`的最后一个字符相同。

如果出现以上情形，我们不需要进行任何编辑操作，开销为0。如：

```
"ACC", "ABC" => "AC", "AB"
cost = 0
```

情形3：子串`str1`和`str2`的最后一个字符不同。

在这种情况下，我们需要计算出以下三种操作的最小开销：

1. 把`str2`的最后一个字符插入到`str1`末尾。
2. 删除`str1`的最后一个字符。
3. 把`str1`的最后一个字符替换成`str2`的最后一个字符。

状态转移方程：

$$T(i,j)=\begin{cases}\max(i,j), & \min(i,j)=0\\ T(i-1,j-1), & str_1(i-1)=str_2(j-1)\\ \min\{T(i-1,j)+1,T(i,j-1)+1,T(i-1,j-1)+2\}, & str_1(i-1)\neq str_2(j-1)\end{cases}$$

3 编程实现

版本一：动态规划

```
# Function to find Minimum Edit Distance between str1 and str2
# m and n are the number of characters in str1 and str2 respectively
def med_dp(str1, str2, m, n):
    # base case: empty strings (case 1)
    if m == 0:
        return n
    if n == 0:
        return m
    # if last characters of the strings match (case 2)
    cost = 0 if (str1[m - 1] == str2[n - 1]) else 2
    return min(med_dp(str1, str2, m - 1, n) + 1,            # deletion (case
3a))
               med_dp(str1, str2, m, n - 1) + 1,            # insertion (case
3b))
               med_dp(str1, str2, m - 1, n - 1) + cost)  # substitution (case 2
+ 3c)
```

版本二：记忆化动态规划

```
# Memoized version of MED
def med_memoized_dp(str1, str2):
    m, n = len(str1), len(str2)
    T = [[0 for x in range(n + 1)] for y in range(m + 1)]
    for i in range(1, m + 1):
        T[i][0] = i                  # (case 1)
    for j in range(1, n + 1):
        T[0][j] = j                  # (case 1)
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:  # (case 2)
                cost = 0                # (case 2)
            else:
                cost = 2                # (case 3c)
            T[i][j] = min(T[i - 1][j] + 1,        # deletion (case 3b)
                          T[i][j - 1] + 1,        # insertion (case 3a)
                          T[i - 1][j - 1] + cost)   # replace (case 2 + 3c)
    return T[m][n], T
```

辅助函数：

```
# Backtrack path along T
def med_backtrack(T):
    path = {}
    diff = [[-1, 0], [0, -1], [-1, -1]]
    i = len(T) - 1
    j = len(T[0]) - 1
    while i!=0 or j!=0:
```

```
        vals = [T[i+diff[0][0]][j+diff[0][1]],
                T[i+diff[1][0]][j+diff[1][1]],
                T[i+diff[2][0]][j+diff[2][1]]]
        min_val = min(vals)
        min_idx = vals.index(min_val)
        i += diff[min_idx][0]
        j += diff[min_idx][1]
        path[(i, j)] = [i-diff[min_idx][0], j-diff[min_idx][1]]
    return path

# Output each step of the shortest path
def output_steps(T, str1, str2):
    path = med_backtrack(T)
    i = 0
    j = 0
    cnt = 0
    prev = str1
    print("Steps:")
    while i!=len(T)-1 or j!=len(T[0])-1:
        next_i = path[(i, j)][0]
        next_j = path[(i, j)][1]
        if T[next_i][next_j]!=T[i][j]:
            str = str2[:next_j] + str1[next_i:]
            cnt += 1
            if next_i-i==1 and next_j-j==0:
                print('Step {} delete "{}": "{}" => "{}"'.format(cnt, str1[i],
prev, str))
            if next_i-i==0 and next_j-j==1:
                print('Step {} insert "{}": "{}" => "{}"'.format(cnt, str2[j],
prev, str))
            if next_i-i==1 and next_j-j==1:
                print('Step {} replace "{}" with "{}": "{}" => "
{}"'.format(cnt,str1[i], str2[j], prev, str))
            prev = str
        i, j = next_i, next_j
```

运行方式：

使用命令行进入文件所在路径，运行下面的任意一条命令：

```
# 使用动态规划的方法，直接输出str1和str2两个字符串的MED
python med.py -med [str1] [str2]
# 使用记忆化动态规划的方法，输出str1和str2两个字符串的MED，并且给出变换过程
python med.py -step [str1] [str2]
```

运行结果示例：

```
Input: "maximum" "minimize"
MED: 9
Table:
[[0, 1, 2, 3, 4, 5, 6, 7, 8], [1, 0, 1, 2, 3, 4, 5, 6, 7], [2, 1, 2, 3, 4, 5, 6,
7, 8], [3, 2, 3, 4, 5, 6, 7, 8, 9], [4, 3, 2, 3, 4, 5, 6, 7, 8], [5, 4, 3, 4, 5,
4, 5, 6, 7], [6, 5, 4, 5, 6, 5, 6, 7, 8], [7, 6, 5, 6, 7, 6, 7, 8, 9]]
Steps:
Step 1 delete "a": "maximum" => "mximum"
Step 2 delete "x": "mximum" => "mimum"
Step 3 insert "n": "mimum" => "minmum"
Step 4 insert "i": "minmum" => "minimum"
Step 5 insert "i": "minimum" => "minimium"
Step 6 replace "u" with "z": "minimium" => "minimizm"
Step 7 replace "m" with "e": "minimizm" => "minimize"
```

4 评估模型

时间复杂度：

版本一：$O(3^n)$，当字符串较长时，将会非常耗时。

版本二：$O(mn)$， 运行时间适中。

空间复杂度：

版本一：$O(1)$，不占用额外空间。

版本二：$O(mn)$，需要存储二维数组T。

版本二的记忆化递归还可以继续优化，只需原来二维数组的前两行就可以计算出MED，但是这样将导致无法回溯出路径。