拼音输入法[1]

计科221董自经 19222126

1. 匹配规则
2. 传统的拼音匹配规则一般是以字的起始字母为拼音的一个分词点去匹配这个字或者词。
3. 除了最后一个字没有全写、缩写的要求以外，不允许中间的字有的输入为全写，有的为缩写，这样会增加汉字匹配的难度，现阶段下这种匹配并不是特别精确，用户自己再查找时也会浪费大量时间，而且不能确保能够找到最佳答案。
4. 多音字在输入时只需要把对应的拼音输入即可，仍需要遵循上述两条规则
5. 实现步骤
6. 拼音分词

需要先对输入的拼音字符串进行合法性检查，并进行分词的处理。

1. 主要问题

元音字母是否需要识别为分词；比如“qingan“，当被识别为”qin‘gan“时，可以输出“情感”，而当被识别为“qing’an“时，会输出“请安‘。

输入是否完整，这主要体现为前后鼻音的区别；比如输入“yin‘gou’bi“，这时算法会自动将”yin“补全为”ying“，从而正确输出“鹰钩鼻”。

输入是否全部是首字母。比如输入“sda“，因为”da“在末尾，所以有多重识别可能，有可能全部是首字母从而输出”手抖啊“，也有可能”da“就是一个整体而只输出”手抖“

1. 关键算法—word-break

word-break算法实现了将一个给定的字符串str和字典dict，并根据dict找出对str拆分的所有可能

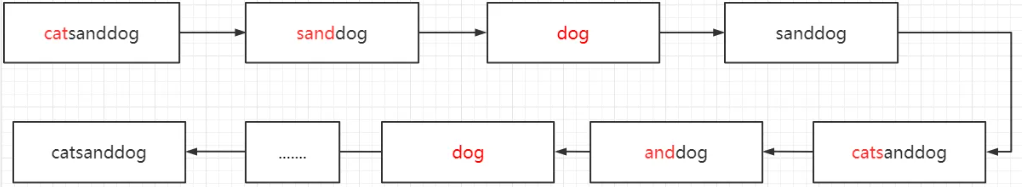
算法描述：从首个字符i开始查找是否在字典中存在以该字母为首字符的单词，若字典中没有该字符，则i + 1再次查找，如果字典中有，则记录该词并对i进行加n，对剩余的字符串重复该操作，如果没有剩余的字符串，完成查找并将记录的词存储到solutions中。回溯到上一次成功的状态，字符i + 1继续重复。退出条件为将全部字符串和字典进行匹配，即i + len(str)

e.g.

s = ‘catsanddog’

dict = [‘cat’,’cats’,’and’,’sand’,’dog’]

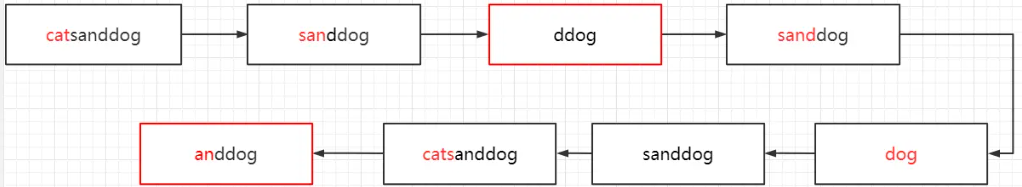
word\_break(s,dict) #结果：[‘cat sand dog’, ‘cats and dog’]



从第一个字符开始，从字典中找到匹配的项目，先找到了cat，之后去掉前面的cat再往后找，匹配到了sand，再往后匹配，找到dog，此时字符串到了末尾，结果有效，存入solutions，回溯到上一次的sand，没有相应的匹配结果，再往上回溯到cat，匹配到cats，之后再次进行匹配，如此操作得到了最终的两个结果

1. 算法优化—剪枝

将已经确认在字典中没有响应结果的字符i的序号保存，下次遇到之后直接跳过从而节省时间，可以另外用一个数组空间来记录该位置。



图中两个红色框的部分没因为ddog在第一次匹配的时候就已经被记录在字典中没有结果，后续再次遇到时直接跳过了。

1. 算法在拼音中的应用

需要有一个完整的拼音字典，在应用中，最复杂的是判断输入是否完整，是否需要补齐剩下的拼音

1. 中文转拼音
2. 需要建立一个拼音字典，原始的字典结构中大概有10000条数据对应了近10000个常见的汉字
3. 之后要实现一个函数，使得其能输出对应中文的拼音，最简单的做法是将（1）中的字典转换为一个map，使得汉字为map中的key，拼音为结果，之后就可以通过汉字查找相应的拼音。
4. 将分词结果和拼音进行匹配

将得到的1.中得到的单词和在2.中得到的拼音进行匹配，形成一个相应的暂时的数据字典，从而确保所得结果的正确性。

字符串编辑距离[2]

计科221董自经 19222126

1. 简介

字符串编辑距离是由俄国数学家Levenshtein发明的，因此又被称为Levenshtein Distence，是指两个字符串之间，有一个转换成灵异的所需要的最少编辑操作次数，这些操作包括替换、插入、删除

1. 算法描述
2. 输入：两个单词word1,word 2
3. 输出：输出需要将第一个单词变为第二个单词所需要的改变次数
4. dp数组
   1. 数组含义

使用数组dp存储相应的编辑距离，dp[i][j] 表示以下标i - 1为结尾的字符串word1，和以下标j - 1为结尾的字符串word2，最小编辑距离为dp[i][j]

* 1. dp数组初始化

a）中之所以用i，j表示word1中下标为i - 1 ，word2中下标为j – 1的字符是因为字符串可能为空，若word2为空，则word1需要进行的操作为全部删除，即dp[i][0] = i。同理dp[0][j] = j

1. word1[i -1] == word2[j - 1]，说明当前位置两个单词中的字母是一致的，不需要进行操作，所以 dp[i][j] = dp[i - 1][j - 1] + 0 = dp[i - 1][j - 1]
2. word1[i - 1] != word2[j - 1]
   1. word1删除一个元素

也就是word1中下标为i – 2的字符与word2中下标为j – 1的字符进行最小编辑距离计算，即dp[i][j] = dp[i - 1][j] + 1

* 1. word2删除一个元素

word2删除元素也就是word1新添加一个元素，但是由于是统计操作次数，不需要修改，所以统一为删除可以减少代码量。也就是word1中下标为i - 1的字符与word2中下标为j - 2的字符进行最小编辑距离计算，即dp[i][j] = dp[i][j -1] + 1

* 1. word1[i - 1]替换为word2[j - 1]

将word1中下标为i - 1 的字符替换为word2中下标为j - 1的字符，此时下标并没有变化，即dp[i][j] = dp[i - 1][j - 1] + 1

* 1. dp[i][j] = min({dp[i][j] = dp[i - 1][j], dp[i][j] = dp[i][j -1], dp[i][j] = dp[i - 1][j - 1] }) +1
  2. 从而确定了四个递推公式

dp[i][j] = dp[i - 1][j - 1]

dp[i][j] = dp[i - 1][j - 1] + 1

dp[i][j] = dp[i][j - 1] + 1

dp[i][j] = dp[i - 1][j] + 1

因为是统计将第一个单词变为第二个单词所需要的改变次数，所以对dp[i][j]的遍历必须是从左到右，从上到下的顺序，这样确保了依赖性、无后效性以及效率。

依赖性：为了计算dp[i][j]，需要知道dp[i-1][j-1]（左上）、dp[i-1][j]（上方）和dp[i][j-1]（左方）的值。如果不按从左到右、从上到下的顺序遍历，可能会尝试在尚未计算这些依赖值的情况下进行访问，这会导致错误的结果。

无后效性：动态规划的一个重要性质是无后效性，即“未来与过去无关，只与现在有关”。在这个问题中，一旦计算了dp[i][j]，就不会再改变，并且值仅依赖于其左方、上方和左上方的值。因此，从左到右、从上到下的遍历顺序确保了当计算dp[i][j]时，其所有依赖项都已经被正确地计算。

效率：虽然理论上可能存在其他遍历顺序也能得出正确结果（例如以某种方式确保在计算dp[i][j]之前其所有依赖项都已经被计算），但从左到右、从上到下的顺序是最简单和直观的，并且很容易实现为嵌套的循环。

* 1. 最后返回dp[i][j]

1. 分析算法

假设word1中有m个字符，word2中有n个字符

1. 时间复杂度

算法中对dp数组的操作只有增、删、改，没有其他，所以时间复杂度主要取决于对于两个字符数组的遍历，算法的时间复杂度就为O(m\*n)

1. 空间复杂度

空间复杂度主要由临时数组dp决定，这个二维数组的大小为(m+1)\*(n+1)，所以需要m\*n的空间来存储，即算法的空间复杂度为O(m\*n)

1. 算法优化的可能途径—剪枝操作

在计算Levenshtein 距离时，可以通过剪枝操作来跳过一些不必要的计算。首先，两个字符串的最小编辑距离有一个最大值，即两个字符串中较长字符串的长度。因此，如果计算过程中的某个值已经大于这个最大值，那么可以跳过该位置的计算。其次，两个字符串的最小编辑距离肯定是小于字符串长度的，如果大于字符串长度也可以直接跳过。

1. 总结

Levenshtein Distance提供了一种直观的方式来量化两个字符串之间的相似性。通过计算将一个字符串转换为另一个字符串所需的最少编辑操作次数，我们可以得到两个字符串之间差异程度的直接度量。这使得它在许多应用中都非常有用，如拼写检查、自然语言处理、信息检索等。

但它也存在一些局限性。它只考虑了单字符编辑操作，而没有考虑更复杂的操作，如子串的插入、删除或替换。这可能导致在某些情况下，Levenshtein Distance无法准确地反映两个字符串之间的实际差异。其次，Levenshtein Distance对字符串的长度敏感，即较长的字符串之间即使只有很小的差异也可能导致较大的Levenshtein Distance值。

参考：

[1]https://zhuanlan.zhihu.com/p/141651129

[2]https://blog.csdn.net/Mwwwwwwww/article/details/121583207