第十二届“认证杯”数学中国

数学建模网络挑战赛
承　诺　书

我们仔细阅读了第十二届“认证杯”数学中国数学建模网络挑战赛的竞赛规则。

我们完全明白，在竞赛开始后参赛队员不能以任何方式（包括电话、电子邮件、网上咨询等）与队外的任何人（包括指导教师）研究、讨论与赛题有关的问题。

我们知道，抄袭别人的成果是违反竞赛规则的，如果引用别人的成果或其他公开的资料（包括网上查到的资料），必须按照规定的参考文献的表述方式在正文引用处和参考文献中明确列出。

我们郑重承诺，严格遵守竞赛规则，以保证竞赛的公正、公平性。如有违反竞赛规则的行为，我们接受相应处理结果。

我们允许数学中国网站(www.madio.net)公布论文，以供网友之间学习交流，数学中国网站以非商业目的的论文交流不需要提前取得我们的同意。

我们的参赛队号为：4466

参赛队员　(签名)　：

队员 1：何东清

队员 2：马壮壮

队员 3：杨壮

参赛队教练员 (签名)：　指导教师组

参赛队伍组别（例如本科组）：本科组

第十二届“认证杯”数学中国

数学建模网络挑战赛
编 号 专 用 页

参赛队伍的参赛队号：（请各个参赛队提前填写好）：

#4466

竞赛统一编号（由竞赛组委会送至评委团前编号）：

竞赛评阅编号（由竞赛评委团评阅前进行编号）：

2019 年第十二届“认证杯”数学中国
数学建模网络挑战赛第一阶段论文

题　　目　基于精准匹配和模糊匹配的外星语字典解析模型

关 键 词　精准匹配 近似匹配 BM 算法 Levenshtein Distance 算法 gram 索引

摘　　要

本文主要讨论的是字符串的匹配问题，我们利用精确匹配算法和近似匹配算法，分别建立了字符串精确匹配模型和字符串近似匹配模型，通过使用两种匹配算法，可以对文本中的字符串匹配得更加准确可靠，克服了 BM 及其改进算法等精确匹配算法遇到字符的替换，删失和插入给匹配带来的大量缺失的问题，确定出了令人满意的字符串匹配数目。

模型 1：假设无替换错误，文本运用 BM 算法对其进行精确匹配，BM 算法主要采用字符串从右向左匹配的方式，它首先将字母序列和目标文本对其，然后从字母序列的最后一个字母开始与其对应的在目标文本中的字母进行匹配，如果匹配成功，则进行倒数第二位进行匹配，直到完全匹配，从而查询出字母序列的个数，如果在匹配过程中有字母匹配不成功，则根据 BM 算法，字母序列整体向后移动，从而和下一位目标对应字母片段匹配，直到匹配成功，同时，我们在 BM 算法的基础上对其进行了改进，采用 BMH 算法算法再次进行字符串的精确匹配。并将匹配到的字符串输出，对于没有匹配到的字符串归为复杂文本，需要进一步分析。

模型 2：假设有替换错误，则对文本字符串运用 Levenshtein Distance 算法和 gram 索引等算法进行字符串的近似匹配。Levenshtein Distance 算法是指两个字符串之间，由一个转成另一个所需的最少编辑操作次数。在本文中，由于题目中说明可能出现的替换错误最多不超过 4 个，所以，我们利用 Levenshtein Distance 算法对目标文本和字母序列进行相似段计算，这里我们选择字母序列长度为 20 个字母，当计算出来的相似度如果大于等于 80%，则将此目标序列认为我们所需要的字母序列，同时，对于没能匹配的字符串再进行 gram 索引算法的近似匹配，gram 索引的局部最优化近似子串查询算法(GASQ)。我们尽可能得防止字符串匹配的遗漏，同样，对于没有匹配到的字符串归为复杂文本，需要进一步分析。

关键词：字符串 精确匹配 近似匹配 BM 算法 Levenshtein Distance 算法 gram 索引。

参赛队号： #4466

所选题目： B　**题**

参赛密码 _____

(由组委会填写)

英文摘要

This article main discussion is the string matching problem, we use an exact match algorithm and approximate matching algorithm, respectively established string matching model precise and approximate string matching model, by using two kinds of matching algorithm, can be in the text string matching more accurate and reliable, improved algorithm overcomes the BM and its exact match algorithm with characters such as replacement, delete and insert to match a lot of the problem of the missing, identify a satisfactory number of string matching.

Model 1: Assuming no substitution errors, text using BM algorithm carries on the exact match, BM algorithm is mainly with the method of matching a string from right to left, it will first letter sequence and the target text, to its and then from the last letter of alphabet sequence start the corresponding letters in the target text matching, if the match is successful, in the penultimate match, until the match completely, thus the query sequence number of the letters, if you have any letters in the matching process matching is not successful, depending on the BM algorithm, letters backward sequence as a whole, and thereby the next target fragments matching corresponding letters, until the match is successful, at the same time, On the basis of BM algorithm, we improved it, and used BMH algorithm to precisely match the strings again. The matched strings are outputted, and the unmatched strings are classified as complex text, which requires further analysis.

Model 2: if replacement error is assumed, Levenshtein Distance algorithm and gram index algorithm are used to approximate match the text string. The Levenshtein Distance algorithm is the minimum number of edits required to go from one string to the other between two strings. In this article, because that may appear in the title of the replacement error is no more than four, so, we use the Levenshtein algorithm of short of target text and letter sequence similarity computation, here we select alphabetical sequence length of 20, if when calculating the similarity is greater than or equal to 80%, will the target sequence, think of what we need letters at the same time, for the failed to match the string "gramm indexing algorithm approximate matching again," gramm index of local approximate substring query optimization algorithm (GASQ). We do our best to prevent the omission of string matches, as well as to classify unmatched strings as complex text, which requires further analysis.

Key words: string exact match approximate match BM algorithm Levenshtein Distance algorithm gram index.

一、问题重述

1.1 背景：

本题属于模式匹配类建模，字符串匹配是模式匹配中最简单的一个问题，在实际应用中，字符串匹配技术在计算机科学，语义学以及分子生物学等领域也具有相当重要的地位,在以模式匹配为主的网络安全应用方面中也发挥着举足轻重的作用。

1.2 需要解决的问题：

一种未知的语言，现只知道其文字是以 20 个字母构成的。已经获取了许多段由该语言写成的文本，但每段文本只是由字母组成的列，没有标点符号和空格，无法理解其规律及含义。我们希望对这种语言开展研究，有一种思路是设法在不同段文本中搜索共同出现的字母序列的片段。语言学家猜测：如果有的序列片段在每段文本中都会出现，这些片段就很可能具备某种固定的含义(类似词汇或词根),可以以此入手进行进一步的研究。在文本的获取过程中，由于我们记录技术的限制，可能有一些位置出现了记录错误。可能的错误分为如下三种：

1. 删失错误：丢失了某个字母；
2. 插入错误：新增了原本不存在的字母；
3. 替换错误：某个字母被篡改成了其他的字母。

第一阶段问题： 假设我们已经获取了 30 段文本，每段文本的长度都在 5000－8000 个字母之间。我们希望找到的片段的长度在 15－21 个字母之间。 为简单起见，我们假设文本中出现的错误只有替换错误，而且对我们要找的 片段而言，在文本中每次出现时，最多只会出现 4 个字母的替换错误。请设计有效的数学模型，快速而尽可能多地找到符合要求的字母片段，并自行编撰算例来验证算法的效果。

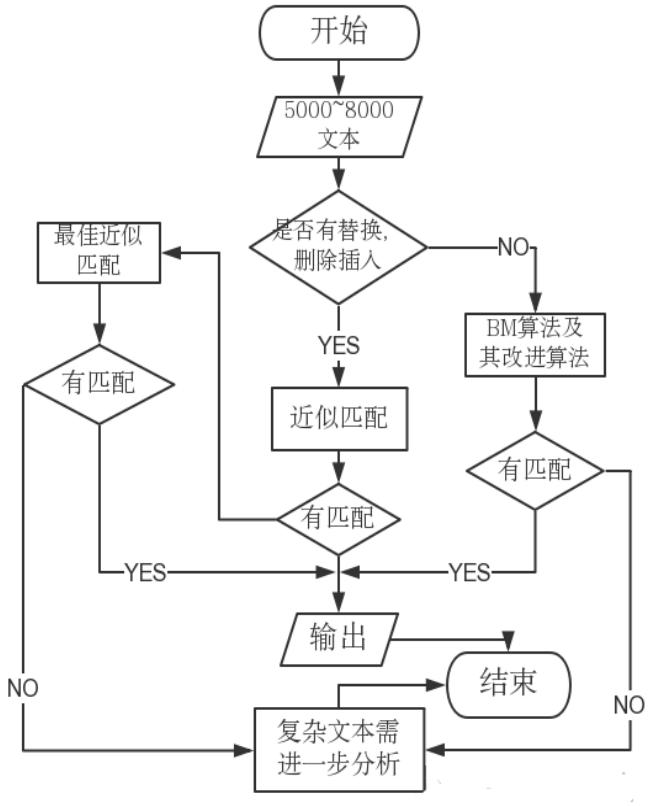
二、问题分析

由于替换，删除和插入错误会对就精准匹配算法 BM 算法产生一定的影响。所以我们把第一阶段分为问题一和问题二。

问题一：假设无替换，删除和插入错误，则对文本运用 BM 算法及其改进算法进行字符串的精准匹配。

问题二：假设有替换，删除和插入错误，则对文本运用 Levenshtein Distance 算法和 gram 索引的模糊匹配算法进行字符串的模糊匹配。

通过将阶段一的问题进一步划分为问题一和问题二,可以对文本中的字符串匹配得更加快速，准确，可靠，同时克服了 BM 及其改进算法等精确匹配算法遇到字符的替换，删失和插入给匹配带来的大量缺失的问题。



问题分析思维图表 1

三、模型假设

（1）文本的字母的范围为 A-Z；
（2）不考虑单个片段替换错误出现 4 次以上的错误；
（3）仅考虑记录时的替换错误，不考虑删失错误和插入错误。
（4）每次查询的目标字母片段只有一个
（5）文本不会出现破损，每一个字符串都可明确辨识；

四、符号说明

P(Pattern)	模式字符串
m	模式字符串的长度
T(Text)	文本字符串
BC	坏字符
GS	好后缀
Skip(x)	P 右移的距离
Shift(j)	P 右移的距离

表 四-1　符号说明

五、模型建立与求解

5.1 对于无替换，删失和插入错误的文本

5.1.1 对于问题一的分析

当无替换，删失和插入错误时，直接利用 BM 算法对字母文本进行匹配，将和目标片段相同的片段找出，并记录次数。

5.1.2 建立 BM 匹配模型

5.1.2.1BM（Boyer-Moore）算法：

BM 算法是一种精确字符串匹配算法。

BM 算法主要采用字符串从右向左匹配的方式，同时使用到了两种启发式规则，即坏字符规则和好后缀规则 ，来决定目标字母片段向右跳跃的距离。

BM 算法的基本流程:

设文本串 T，模式串为 P。首先将 T 与 P 进行左对齐，然后进行从右向左比较 ，如下图所示:

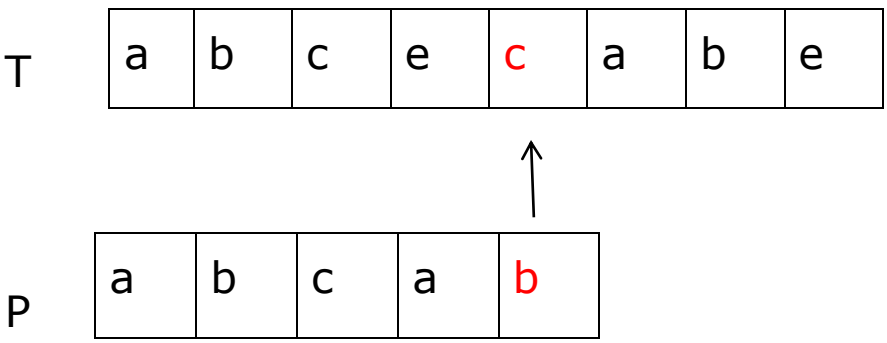


图 五-1

当某趟比较不匹配时，BM 算法就采用两条启发式规则，即坏字符规则和好后缀规则 ，来计算模式串向右移动的距离，直到整个匹配过程的结束。

以下介绍一下坏字符规则 和好后缀规则 。

首先，诠释一下坏字符和好后缀的概念。

请看下图：

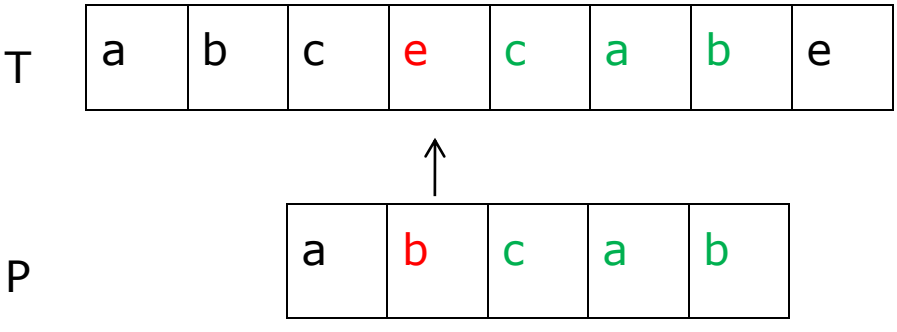


图 五-2

图中，第一个不匹配的字符（红色部分）为坏字符，已匹配部分（绿色）为好后缀。

（1）坏字符规则（Bad Character）：

在 BM 算法从右向左扫描的过程中，若发现某个字符 x 不匹配，则按如下两种情况讨论：

i. 如果字符 x 在模式 P 中没有出现，那么从字符 x 开始的 m 个文本显然不可能与 P 匹配成功，直接全部跳过该区域即可。

ii. 如果 x 在模式 P 中出现，则以该字符进行对齐。

用数学公式表示，设 Skip(x)为 P 右移的距离，m 为模式串 P 的长度，max(x)为字符 x 在 P 中最右位置。

$$\text{skip（x）}=\begin{cases} \text{m；} \quad x \neq P[j](1 \le j \le m), \text{即x在}P\text{中未出现} \\ \text{m - max（x）} \quad \{k \mid P[k]=x, 1 \le k \le m\}; \text{x在P中出现} \end{cases}$$

公式 1

例 1：

下图红色部分，发生了一次不匹配。

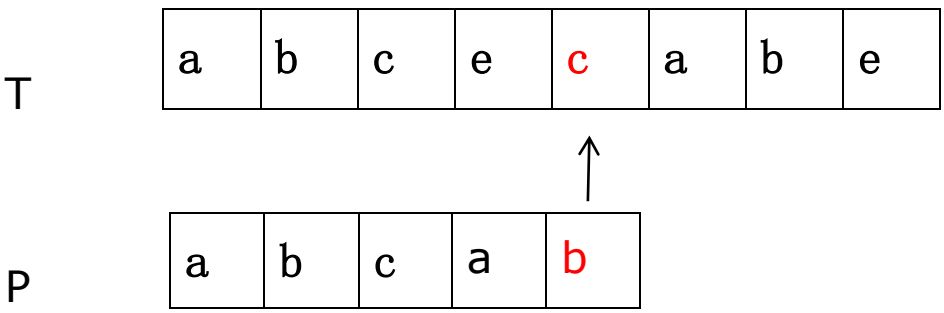


图 五-3

计算移动距离 Skip(c) = 5 - 3 = 2，则 P 向右移动 2 位。

移动后如下图：

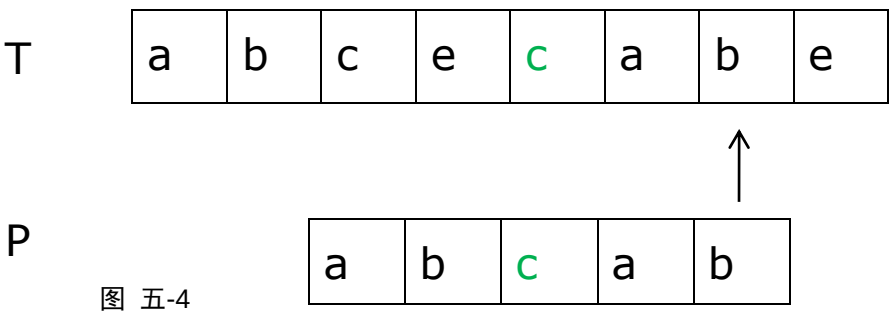


图 五-4

（2）好后缀规则（Good Suffix）：

若发现某个字符不匹配的同时，已有部分字符匹配成功。

用数学公式表示，设 Shift(j)为 P 右移的距离，m 为模式串 P 的长度，j 为当前所匹配的字符位置，s 为 t’与 t 的距离（以上情况 i）或者 x 与 P’’的距离（以上情况 ii）。

$$shift(j)=\min\{s\mid(P[j+1..m]=P[j-s+1..m-s])\&\&(P[j]\neq P[j-s])(j>s),P[s+1..m]=P[1..m](j\leq s)\}$$

以上过程有点抽象，所以我们继续图解。

例 2：

下图中，已匹配部分 cab（绿色）在 P 中再没出现。

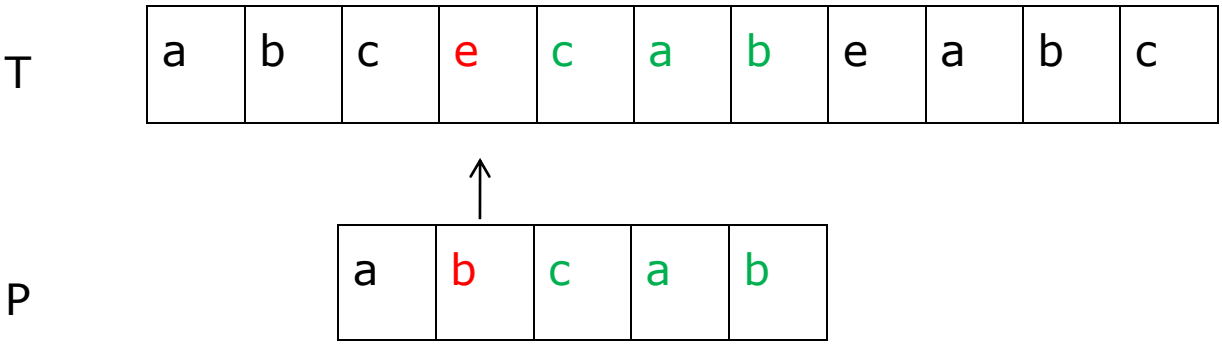


图 五-5

再看下图，其后缀 T’（蓝色）与 P 中前缀 P’（红色）匹配，则将 P’移动到 T’的位置。

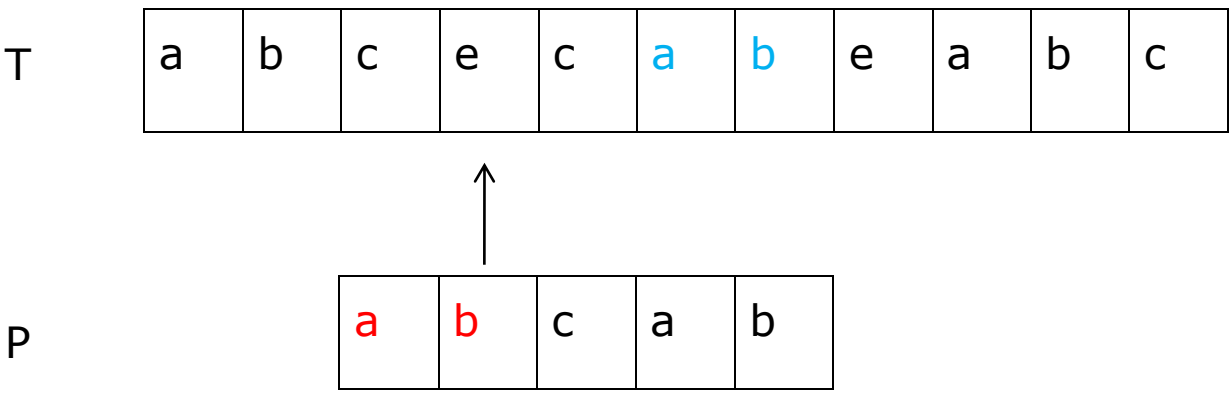


图 五-6

移动后如下图：

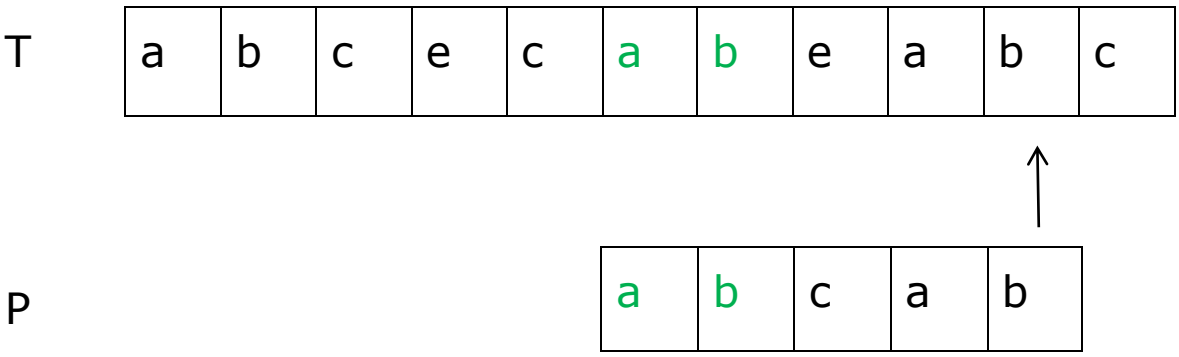


图 五-7

在 BM 算法匹配的过程中，取 SKip(x)与 Shift(j)中的较大者作为跳跃的距离。

BM 算法预处理时间复杂度为 O（m+s），空间复杂度为 O(s)，s 是与 P, T 相关的有限字符集长度，搜索阶段时间复杂度为 O(m • n)。

最好情况下的时间复杂度为 O(n/m)，最坏情况下时间复杂度为 O(m • n)。

5.1.2.2 BM 改进算法

1. BMH 算法：

算法思想：

bm 算法中的不良字符跳转函数对于小字符集（例如生物信息处理中表示 DNA 分子的字符集）应用而言并不能取得很好的效率，但是如果字符集相对于模式串 P 很大（例如 ASIIC 码），将会取得非常好的效果。正是基于这一点 Horspool 改进了 BM 算法，算法只使用了不良字符跳转表 bmBc[]。

算法描述：

给定主串 T 和模式串 P，返回 P 在 T 中首次出现的位置，如果 P 不存在于 T 中，返回-1。这样的问题就是字符串匹配问题，这里给出 BMH 算法的思想。设主串 T 的长度为 n, 模式串 P 的长度为 m。BMH(Boyer-Moore-Horspool)算法是 BM(Boyer-Moore)算法的一种优化，根据《一种基于 BMH 算法的模式匹配算法》的分析，BMH 算法要优于 BM 算法，BM 算法的思想可以参考字符串匹配的 Boyer-Moore 算法。BM 算法的核心在于两个启发式算法，一个叫做坏字符(bad character)，一个叫做好后缀(good suffix)。BMH 它不再像 BM 算法一样关注失配的字符(好后缀)，它的关注的焦点在于这个坏字符上，根据这个字符在模式串 P 中出现的最后的位置算出偏移长度，否则偏移模式串的长度。

偏移表

在预处理中，计算大小为|Σ||Σ|的偏移表。

$$shift|w| = \begin{cases} m-1-\max\{i < m-1 | P|i| = w\}, & if \quad w \quad is \quad in \quad P[0..m-2] \\ m \qquad\qquad\qquad\qquad\qquad , & Otherwise \end{cases}$$

公式 2

例如： P = “pappar”

m = 6

shift[p] = 6 - 1 - max(p 的位置) = 6 - 1 - 3 = 2

shift[a] = 6 - 1 - max(a 的位置) = 6 - 1 - 4 = 1

算法分析

最坏情况运行时间

- 预处理：O(|Σ||Σ|+m)
- 搜索：O(nm)
- 总计：O(nm)

空间：|Σ||Σ|，和 m 独立

该算法在真实数据集合上非常快

2.字符串匹配——Sunday 算法：

算法思想：Sunday 算法由 Daniel M.Sunday 在 1990 年提出，它的思想跟 BM 算法很相似：

只不过 Sunday 算法是从前往后匹配，在匹配失败时关注的是主串中参加匹配的最末位字符的下一位字符。

如果该字符没有在模式串中出现则直接跳过，即移动位数 = 模式串长度 + 1；

否则，其移动位数 = 模式串长度 - 该字符最右出现的位置(以 0 开始) = 模式串中该字符最右出现的位置到尾部的距离 + 1。

算法例子：

下面举个例子说明下 Sunday 算法。假定现在要在主串”substring searching”中查找模式串”search”。

刚开始时，把模式串与文主串左边对齐：

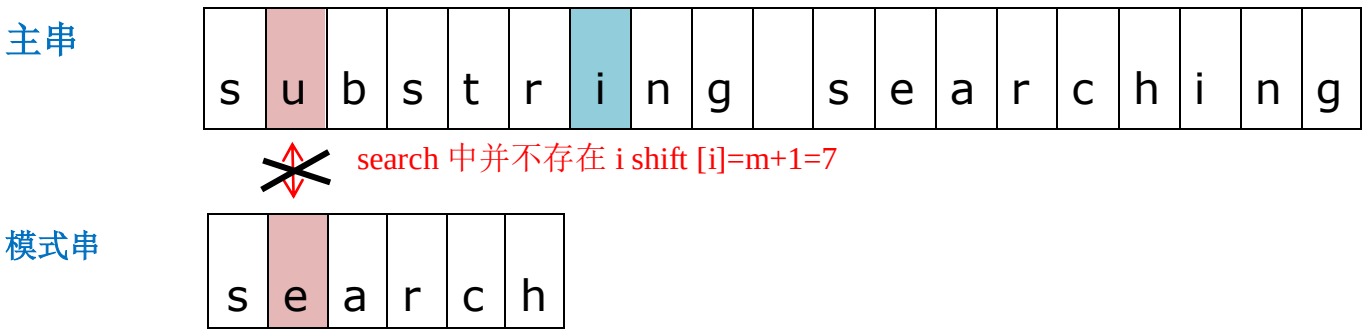
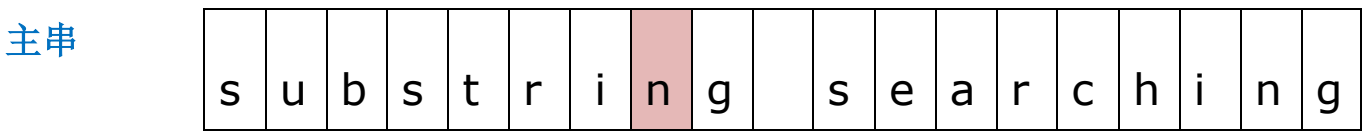


图 五-8

结果发现在第 2 个字符处发现不匹配，不匹配时关注主串中参加匹配的最末位字符的下一位字符，即标粗的字符 i，因为模式串 search 中并不存在 i，所以模式串直接跳过一大片，向右移动位数 = 匹配串长度 + 1 = 6 + 1 = 7，从 i 之后的那个字符（即字符 n）开始下一步的匹配，如下图：



search 中 r 的位置为 3　shift[r]=m-3=3

模式串

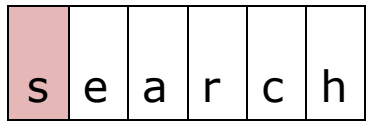
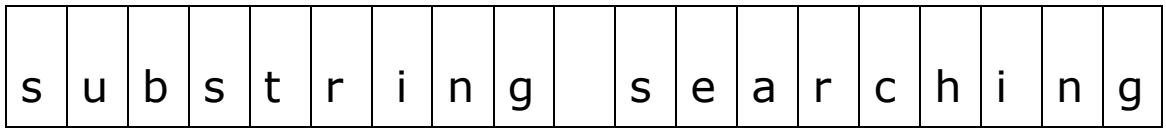


图 五-9

结果第一个字符就不匹配，再看主串中参加匹配的最末位字符的下一位字符，是’r’，它出现在模式串中的倒数第 3 位，于是把模式串向右移动 3 位（m - 3 = 6 - 3 = r 到模式串末尾的距离 + 1 = 2 + 1 =3），使两个’r’对齐，如下：

主串



匹配成功

模式串

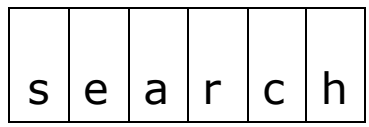


图 五-10

匹配成功。

回顾整个过程，我们只移动了两次模式串就找到了匹配位置，缘于 Sunday 算法每一步的移动量都比较大，效率很高。

偏移表

在预处理中，计算大小为|Σ||Σ|的偏移表。

$$shift|w| = \begin{cases} m - \max\{i < m | P|i| = w\}, & if \quad w \quad is \quad in \quad P[0..m-1] \\ m+1 \quad , & Otherwise \end{cases}$$

公式 3

例如： P = “search”

m = 6

shift[s] = 6 - max(s 的位置) = 6 - 0 = 6

shift[e] = 6 - max(e 的位置) = 6 - 1 = 5

shift[a] = 6 - max(a 的位置) = 6 - 2 = 4

shift[r] = 6 - max(r 的位置) = 6 - 3 = 3

shift[c] = 6 - max(c 的位置) = 6 - 4 = 2

shift[h] = 6 - max(h 的位置) = 6 - 5 = 1

shift[其他] = m + 1 = 6 + 1 = 7

算法分析 2

Sunday 预处理阶段的时间为：O(|Σ||Σ| + m)

最坏情况下时间复杂度为：O(nm)

平均时间复杂度：O(n)

空间复杂度：O(|Σ||Σ|)

5.1.3 模型一求解

生成目标字母程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define STR_LEN 180000//定义随机输出的字符串长度。
#define CHAR_MIN 'A'
#define CHAR_MAX 'Z' //定义输出随机字符串每个字符的最大最小值。
int main()
{
    char str[STR_LEN + 1] = {0};
    int i;

    srand(time(NULL));//通过时间函数设置随机数种子，使得每次运行结果随机。
    for(i = 0; i < STR_LEN; i ++)
    {
        str[i] = rand()%(CHAR_MAX-CHAR_MIN + 1) + CHAR_MIN; //生成要求范围内的随机数。
    }
    printf("%s\n", str);//输出生成的随机数。

    return 0;
    }
```

BM 算法程序：

已整理到附录。

5.2 对于有替换，删失和插入错误的文本

5.2.1 对于有替换，删失和插入错误的文本的分析

当出现替换错误时，此时的精准匹配就起不上什么作用了，已知题中要求，在一个字母序列中，最多可出现四个错误，此时，可以利用模糊匹配来进行字符串的查询，将目标字符串和字母片段进行对比，当他的近似度大于某一个值时，将此值再次进行精准模糊匹配，如果小于某一个值，则直接进行舍弃。

5.2.2 建立近似匹配以及最有近似匹配模型

5.2.2.1Levenshtein Distance 算法的建立

Levenshtein Distance 算法，又叫 Edit Distance 算法，是指两个字符串之间，由一个转成另一个所需的最少编辑操作次数。

算法实现原理图解：

a.首先是有两个字符串,这里写一个简单的 abc 和 abe

b.将字符串想象成下面的结构。

A 为一个标记

	abc	a	b	c
abc	0	1	2	3
a	1	A		
b	2			
c	3			

表 五-11

c.来计算 A 处的值

它的值取决于：左边的 1、上边的 1、左上角的 0。

按照 Levenshtein distance 的意思：

上面的值加 1 ，得到 1+1=2 ，

左面的值加 1 ，得到 1+1=2 ，

左上角的值根据字符是否相同，相同加 0 ，不同加 1 。A 处由于是两个 a 相同，左上角的值加 0 ，得到 0+0=0 。

然后从我们上面计算出来的 2，2，0 三个值中选取最小值，所以 A 处的值为 0 。

d.于是表成为下面的样子

	abc	a	b	c
abe	0	1	2	3
a	1	0		
b	2	B		
e	3			

表 五-12

在 B 处会同样得到三个值，左边计算后为 3 ，上边计算后为 1 ，在 B 处由于对应的字符为 a、b ，不相等，所以左上角应该在当前值的基础上加 1 ，这样得到 1+1=2 ，在（3,1,2）中选出最小的为 B 处的值。

于是

	abc	a	b	c
abe	0	1	2	3
a	1	0		
b	2	1		
e	3	C		

表 五-13

C 处计算后：上面的值为 2 ，左边的值为 4 ，左上角的：a 和 e 不相同，所以加 1 ，即 2+1 ，左上角的为 3 。

在（2,4,3）中取最小的为 C 处的值。

	abc	a	b	c
abe	0	1	2	3
a	1	0		
b	2	1		
e	3	2		

表 五-14

依次得到：

	abc	a	b	c
abe	0	1	2	3
a	1	A=0	D=1	G=2
b	2	B=1	E=0	H=1
e	3	C=2	F=1	I=1

表 五-15

I 处： 表示 abc 和 abe 有 1 个需要编辑的操作（ c 替换成 e ）。这个是需要计算出来的。

同时，也获得一些额外的信息：

A 处： 表示 a 和 a 需要有 0 个操作。字符串一样

B 处： 表示 ab 和 a 需要有 1 个操作。

C 处： 表示 abe 和 a 需要有 2 个操作。

D 处： 表示 a 和 ab 需要有 1 个操作。

E 处： 表示 ab 和 ab 需要有 0 个操作。字符串一样

F 处： 表示 abe 和 ab 需要有 1 个操作。

G 处： 表示 a 和 abc 需要有 2 个操作。

H 处： 表示 ab 和 abc 需要有 1 个操作。

I 处： 表示 abe 和 abc 需要有 1 个操作。

则计算相似度为：

先取两个字符串长度的最大值 maxLen，用 1-（需要操作数 除 maxLen），得到相似度。

例如 abc 和 abe 一个操作，长度为 3 ，所以相似度为 1-1/3=0.666 。

5.2.2.2 支持局部最优化匹配的近似子串查询算法-gram 索引

一个字符串的 gram 是指它特定长度的子串，如“program”的 2-gram 集合为{pr,ro,og,gr,ra,am}。

5.2.3 近似匹配以及最有近似匹配求解

```
#include <stdio.h>
#include <string.h>

#define MIN3(a,b,c) ((a)<(b)?((a)<(c)?(a):(c)):((b)<(c)?(b):(c)))

int cmp_levenshtein( const char *s1, const char *s2 );

int main( void )
{
	char str1[] = "Today is Saturday.";
	char str2[] = "Tomorrow is Sunday.";
	int d = cmp_levenshtein( str1, str2 );
	printf( "edit distance: %d\n", d );
	return 0;
}

int cmp_levenshtein( const char *s1, const char *s2 )
{
	int row = strlen(s1);			/* s1  的长度  */
	int col = strlen(s2);			/* s2  的长度  */

	int mat[row][col];			/* C99 - variable-length array */

	for( int i=0; i<row; ++i ) {		/*  数组的行  */
		for( int j=0; j<col; ++j ) {	/*  数组的列  */
			if( i == 0 ) {
				mat[i][j] = j;		/*  初始化第 1 行为  [ 0 1 2 ... ] */
			}
			else if( j == 0 ) {
				mat[i][j] = i;		/*  初始化第 1 列为  [ 0 1 2 ... ] */
			}
			else {
				int cost = ( s1[i-1] == s2[j-1] ) ? 0 : 1;	/*  记录 s1[i-1]与 s2[j-1]是否相等  */
				mat[i][j] = MIN3( mat[i-1][j  ] + 1,		/*  取三者的最小值  */
				                  mat[i  ][j-1] + 1,
				                  mat[i-1][j-1] + cost);
			}
		}
	}

	return mat[row-1][col-1];
}
```

六、模型检验

我们利用 C 语言先产生了一个 30 行，每一行 6000 个字母的数组，用来充当我们的目标序列，如下：

图 1

目标序列

然后通过 BM 算法对其进行精准匹配

已知我们的字母序列假设为 AMNHGBFDVSGHH,

一下为 BM 算法运行结果图：

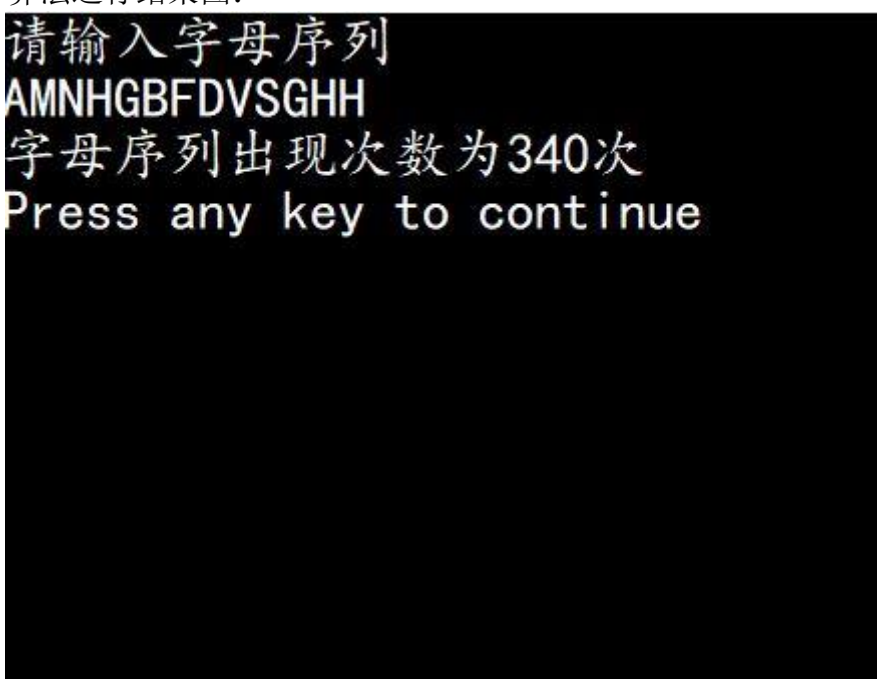


图 2

七、模型评价

7.1 模型的优点

该模型运用 BM 算法求解，该算法是目前应用最为广泛的单模式匹配算法。我们运用 BM 算法的主要特点，在运用该算法进行字符串的匹配时，跳过了很多无效的字符串。通过这种跳跃式的匹配，获得了较高的执行效率。有实验表明，BM 算法的匹配速度大约是 KMP 算法的 4 倍。

7.2 模型的缺点：

其后缀规则在理论上效果好，但在应用中作用很小，反而增加了算法的总的运行时间，影响了算法啊的性能。并且该模型不能自动检测是否有替换，删失和插入错误。

7.3 模型的改进方向：

针对模型的缺点，为进一步缩短算法的匹配时间，提高系统的搜索效率，可以改用 BM 算法的改进算法，如 BMH 算法，BMH 算法，Sunday 算法等。

针对模型不能自动检测是否有替换，删失和插入错误，我们可以重新修改模型的结构，我们可以先运用 BM 及其改进算法对字符串进行精准匹配，而对于没有匹配到的字符串进行模糊匹配。从而直接可以对无错误和有错误的字符串进行匹配，提升了执行效率。

也可用 Bloom Filter 算法实现字符串的精确匹配和模糊匹配。

精确算法：

我们可以创建 1000 个文件，运用哈希函数先将文件 1 的字符串保存在对应的文件中，之后再文件 2 中取元素，通过哈希函数计算出哈希地址，去对应的文件里面找是否有与之相同的字符串。

近似算法：

我们可以使用位图的方法，通过一个函数将一个元素映射成一个位矩阵中的一个点，这样一来，我们只要看看这个点是不是 1 就知道集合里有没有它了。 但是有可能两个字符串对应的整数是一样的，对于这种情况我们可以设置更多的哈希函数，对应更多的地址，这样更加精确。

八、参考文献

[1] 毕智超.字符串模式匹配算法的研究及改进陕西职业技术学院 ， 陕西西安 2013.20

[2]张国平，徐汶东.字符串模式匹配算法的改进 中国石油大学（华东）计算机与通信工程学院 2007.10

[3]范洪博.快速精确字符串匹配算法研究 哈尔滨工业大学 2011.11

[4]邓惠俊.多模式匹配算法的研究 合肥工业大学 计算机技术 2009.10

[5]韩忠庚 数学建模方法及其应用.北京：高等教育出版社，2005

[6]司守奎.数学建模算法与应用(第二版).北京：国防工业出版社，2015

九、附录

1 BM 算法

```
//数组 bmBc 的创建//
void preBmBc(char *x, int m, int bmBc[]) {
    int i;
```

```
    for (i = 0; i < ASIZE; ++i)
        bmBc[i] = m;
    for (i = 0; i < m - 1; ++i)
        bmBc[x[i]] = m - i - 1;
}

//构建 suffix 数组//
void suffixes(char *x, int m, int *suff) {
    int f, g, i;
    suff[m - 1] = m;
    g = m - 1;
    for (i = m - 2; i >= 0; --i) {
        if (i > g && suff[i + m - 1 - f] < i - g)
            suff[i] = suff[i + m - 1 - f];
        else {
            if (i < g)
                g = i;
            f = i;
            while (g >= 0 && x[g] == x[g + m - 1 - f])    //好后缀处理
                --g;
            suff[i] = f - g;
        }
    }
}

//构建 bmGs 数组//
void preBmGs(char *x, int m, int bmGs[]) {
    int i, j, suff[XSIZE];
    suffixes(x, m, suff);
    for (i = 0; i < m; ++i)
        bmGs[i] = m;
    j = 0;
    for (i = m - 1; i >= -1; --i)
        if (i == -1 || suff[i] == i + 1)
            for (; j < m - 1 - i; ++j)
                if (bmGs[j] == m)
                    bmGs[j] = m - 1 - i;
    for (i = 0; i <= m - 2; ++i)
        bmGs[m - 1 - suff[i]] = m - 1 - i;
}

//  再来重写一遍 BM 算法//
void BM(char *x, int m, char *y, int n) {
    int i, j, bmGs[XSIZE], bmBc[ASIZE];

    /* Preprocessing */
    preBmGs(x, m, bmGs);
    preBmBc(x, m, bmBc);

    j = 0;
    while (j <= n - m) {
        for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
        if (i < 0) {
            OUTPUT(j);
            j += bmGs[0];
        }
```

```
        else
            j += MAX(bmGs[i], bmBc[y[i + j]] - m + 1 + i);
    }
}
```

2 BMH 算法

```
void HORSPOOL(char *x, int m, char *y, int n) {
    int j, bmBc[ASIZE];
    char c;
    /* Preprocessing */
    preBmBc(x, m, bmBc);
    /* Searching */
    j = 0;
    while (j <= n - m) {
        c = y[j + m - 1];
        if (x[m - 1] == c && memcmp(x, y + j, m - 1) == 0)
            OUTPUT(j);
        j += bmBc[c];
    }
}
```

3 BMHS 算法

```
void preQsBc(char *x, int m, int qsBc[]) {
    int i;

    for (i = 0; i < ASIZE; ++i)
        qsBc[i] = m + 1;
    for (i = 0; i < m; ++i)
        qsBc[x[i]] = m - i;
}
void QS(char *x, int m, char *y, int n) {
    int j, qsBc[ASIZE];
    /* Preprocessing */
    preQsBc(x, m, qsBc);
    /* Searching */
    j = 0;
    while (j <= n - m) {
        if (memcmp(x, y + j, m) == 0)
            OUTPUT(j);
        j += qsBc[y[j + m]];                    /* shift */
    }
}
```

4 . smith 算法

```
void SMITH(char *x, int m, char *y, int n) {
    int j, bmBc[ASIZE], qsBc[ASIZE];

    /* Preprocessing */
    preBmBc(x, m, bmBc);
    preQsBc(x, m, qsBc);

    /* Searching */
    j = 0;
    while (j<= n - m) {
        if (memcmp(x, y + j, m) == 0)
```

```
            OUTPUT(j);
        j += MAX(bmBc[y[j + m - 1]], qsBc[y[j + m]]);
    }
}
```

5. 近似匹配算法

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <stack>
#include <limits>
using namespace std;
int main()
{
    ifstream fin("in1.txt");
    ofstream out("out.txt");
    stringstream    ss;
    string keyword = "1.4.3  怎样编译 Perl？ ";
    ss << fin.rdbuf();
    string text = ss.str();
    vector<int> c1;    //  距离矩阵第 c[j-1]列
    vector<int> c2;    //  距离矩阵第 c[j]列
    c1.resize(keyword.size()+1);
    c2.resize(keyword.size()+1);
    for(int i = 0; i <= keyword.size(); i++)
        c1[i] = i;
    c2[0] = 0;
    int minStep =    numeric_limits<int>::max();
    int matchpos;
    for (int i = 1; i <= text.size(); i++)
    {
        for(int j = 1; j <= keyword.size(); j++)
        {
            c2[j] = c1[j-1] + (text[i-1] == keyword[j-1] ? 0 : 1);
            if (c2[j] > c1[j] + 1)
                c2[j] = c1[j] + 1;
            if (c2[j] > c2[j-1] + 1)
                c2[j] = c2[j-1] + 1;
        }
        for(int j = 1; j <= keyword.size(); j++)
            c1[j] = c2[j];
        if (minStep >= c2[keyword.size()]) //  取最后一个最近似的子串
        {
            minStep = c2[keyword.size()];
            matchpos = i;
        }
    }
    cout << text << endl;
    cout << "keyword:" << keyword    << endl;
    cout << "match text: " << text.substr(matchpos - keyword.size(), keyword.size()) << endl;
    cout << "match postion:" << matchpos << endl;
    getchar();
    return 0;
```

```
}
----------------------------------字母表---------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define STR_LEN 180000//定义随机输出的字符串长度。
#define CHAR_MIN 'A'
#define CHAR_MAX 'Z' //定义输出随机字符串每个字符的最大最小值。
int main()
{
	char str[STR_LEN + 1] = {0};
	int i;

	srand(time(NULL));//通过时间函数设置随机数种子，使得每次运行结果随机。
	for(i = 0; i < STR_LEN; i ++)
	{
		str[i] = rand()%(CHAR_MAX-CHAR_MIN + 1) + CHAR_MIN; //生成要求范围内的随机数。
	}
	printf("%s\n", str);//输出生成的随机数。

	return 0;
}----------------------------------------BM 精准算法-------------------------------------------------
#include <stdio.h>
#include <string.h>

//生成 skip 数组，即 delta1 数组
int *make_skip(char *ptrn, int plen)
{
	int *skip = (int *)malloc(256 * sizeof(int));
	int *sptr = skip + 256;

	if (skip == NULL)
		fprintf(stderr, "malloc failed!");

	while (sptr-- != skip)
		*sptr = plen + 1;

	while (plen != 0)
		skip[(unsigned char)*ptrn++] = plen--;

	return skip;
}

//生成 shift 数组，即 delta2 数组
int *make_shift(char *ptrn, int plen)
{
	int *shift = (int *)malloc(plen * sizeof(int));
	int *sptr = shift + plen - 1;
	char *pptr = ptrn + plen - 1;
	char c;

	if (shift == NULL)
		fprintf(stderr, "malloc failed!");

		c = ptrn[plen - 1];
```

```
            *sptr = 1;

        while (sptr -- != shift)
        {
            char *p1 = ptrn + plen - 2, *p2, *p3;

        do
        {
            while (p1 >= ptrn && *p1-- != c);

                p2 = ptrn + plen - 2;
                p3 = p1;

        while (p3 >= ptrn && *p3-- == *p2-- && p2 >=pptr);
        }
        while (p3 >= ptrn && p2 >= pptr);

            *sptr = shift + plen - sptr + p2 - p3;

            pptr--;
        }
        return shift;
}

//搜索函数
int mSearch(char *buf, int blen, char *ptrn, int plen, int *skip, int *shift)
{
int b_idx = plen;

if (plen == 0)
return 1;

while (b_idx <= blen)
{
int p_idx = plen, skip_stride, shift_stride;

while (buf[--b_idx] == ptrn[--p_idx])
{
if (b_idx < 0)
return 0;

if (p_idx == 0)
{
return 1;
}
}

skip_stride = skip[(unsigned char)buf[b_idx]];
shift_stride = shift[p_idx];

b_idx += (skip_stride > shift_stride) ? skip_stride : shift_stride;
}

return 0;
}

int main()
```

```
{
char str[100] = "fABCDEaecabcxxdefeabcxxxabcdwaw";
char pattern[10] = "ABCDE";//abcxxxabc
int *skip, *shift, i;

skip = make_skip(pattern, strlen(pattern));
shift = make_shift(pattern, strlen(pattern));

if (!mSearch(str, strlen(str), pattern, strlen(pattern), skip, shift))
printf("The string \"%s\" doesn't contain string \"%s\"\n", str, pattern);
else
printf("The string \"%s\" does contain string \"%s\"\n", str, pattern);

return 0;
}
----------------------------------字符串相似度算法——Levenshtein Distance 算法----------------
#include <stdio.h>
#include <string.h>

#define MIN3(a,b,c) ((a)<(b)?((a)<(c)?(a):(c)):((b)<(c)?(b):(c)))

int cmp_levenshtein( const char *s1, const char *s2 );

int main( void )
{
	char str1[] = "Today is Saturday.";
	char str2[] = "Tomorrow is Sunday.";
	int d = cmp_levenshtein( str1, str2 );
	printf( "edit distance: %d\n", d );
	return 0;
}

int cmp_levenshtein( const char *s1, const char *s2 )
{
	int row = strlen(s1);			/* s1  的长度  */
	int col = strlen(s2);			/* s2  的长度  */

	int mat[row][col];			/* C99 - variable-length array */

	for( int i=0; i<row; ++i ) {		/*  数组的行  */
		for( int j=0; j<col; ++j ) {	/*  数组的列  */
			if( i == 0 ) {
				mat[i][j] = j;		/*  初始化第 1 行为  [ 0 1 2 ... ] */
			}
			else if( j == 0 ) {
				mat[i][j] = i;		/*  初始化第 1 列为  [ 0 1 2 ... ] */
			}
			else {
				int cost = ( s1[i-1] == s2[j-1] ) ? 0 : 1;	/*  记录 s1[i-1]与 s2[j-1]是否相等  */
				mat[i][j] = MIN3( mat[i-1][j  ] + 1,		/*  取三者的最小值  */
						  mat[i  ][j-1] + 1,
						  mat[i-1][j-1] + cost);
			}
		}
	}
```

```
    return mat[row-1][col-1];
}
```