**论文编号：**

**多种字符串匹配算法的优化与创新**

**摘要**: 字符串匹配技术在许多领域里被广泛应用。分析了KMP、BM字符串匹配思想方法和预处理模式串的算法，并提供了代码实现;对KMP算法模式串的预处理进行了优化；对字符串模糊匹配提出预处理目标串的NP算法，提供了新思路；对多模式串的匹配运用了字典树的数据结构，并提出合理融合DP思想的算法。

**关键词:** 字符串；模式匹配；KMP；BM；模糊匹配；字典树；DP

**Improvement And Innovation Of Multiple Algorithms Of String Matching**

**Abstract**: The technology of string matching is applied aboard in many fields.This paper analyzes KMP,Boyer-Moore and their processing algorithms,and provides the implementation codes. And it improves pattern-string-processing algorithm of KMP algorithm. Additionally,An algorithm named NP is provided,featuring on target-string-processing.The data structure of Trie is used for the matching of multi-mode strings, and an algorithm mixed with DP is proposed.

**Key Words**: Character string;Pattern matching;KMP;BM;Fuzzy matching;Trie;DP

—————————————

# 1 引 言

字符串模式匹配分为精确匹配与模糊匹配，是计算机科学的重要组成部分，蕴含众多信息、科学理论、算法思想以及算法技巧，其应用渗透了信息技术的各个领域，甚至发展成了一门学科。

单字符串精确匹配作为信息检索的基础，是信息检索的瓶颈技术，直接影响到信息检索的检索方式、检索功能、检索效果、用户界面等。随着网络信息大众化的快速发展，用户对信息检索提出了更高要求，功能上要求提高查全率、查准率以及精确定位，操作上要求简单、灵活、快捷。

模糊字符串匹配方法主要包括容错纠错匹配、最大匹配、相似匹配等。模糊字符串匹配允许出现有限的错误，多数情况下通过按顺序出现规定的字符、相似度或距离等方式进行约束，返回匹配结果以及匹配位置。

多模式字符串匹配算法在很领域都有重要应用,如拼写检查、语言翻译、数据压缩、搜索引擎、网络入侵检测等。研究高效的多模式匹配算法具有非常重要的理论和现实意义。

# 2 优化算法：KMP算法及预处理模式串的优化

## 2.1 KMP算法概述

这是一种改进的模式匹配算法。字符串的模式匹配问题描述如下:设有两个字符串T和P，若打算在串T中查找否有与串 P相等的子串，则称串T为目标(Target),称P为模式(Pattern),并称查找模式串在目标串中的匹配位置的运算为模式匹配(Pattern matching)。字符串匹配对众多应用具有重要意义，如入侵检测等。对于字符串匹配，朴素算法的低下效率是难以被接受的，达到了O（t.length()\*p.length()）。而 KMP算法则是一种无回溯的算法,大大提高了在出现“部分匹配”情况下的模式匹配效率。其改进在于：每当一次匹配过程中出现字符不等时，无需回退目标串T的当前字符，而是根据已经预处理模式串P得到的失效函数值右移模式串P，继续进行比较，时间复杂度为O(t.length()+p.length())。

失效函数f[j]的含义阐释如下：（若无特别说明，本文默认规定字符串的首字符下标为0）当t[i]≠p[j]时，显然已经确定p[0]p[1]…p[j-1]==t[i-j+1]t[i-j+2]…t[i]。如果存在f[j]==k（0≤k<j）,使得p[0]p[1]…p[k]==p[j-k]p[i-k+1]…p[j],那么直接右移模式串P，使得p[0]p[1]…p[k]覆盖了p[j-k]p[i-k+1]…p[j]在移动之前的位置，那么此时这些位置与目标串T的对应位置也显然匹配，因此只需要接着比较字符即可。

确定失效函数的代码如下：

void faiArrayl(int f[], string p){

f[0] = -1;

int i = f[0];

int j = 1;

while (j < p.length()){

if (p[i + 1] == p[j]){

f[j] = i + 1;

j++; i++;

}

else{

if (i == -1){ {

f[j] = -1;

j++;

}

else {

i = f[i];

}

}

}

}

KMP基于失效函数数组fail[]的匹配算法如下：

int KMP(string t, string p, int fail[]){

int i = 0, j = 0;

while (i < t.length() && j < p.length()) {

if (t[i] == p[j]){ {

i++;

j++;

}

else {

if (j == 0) {

i++;

}

else{

j = fail[j - 1] + 1;

}

}

}

if (j < p.length()){

return -1;

}

else{

return i-j;

}

}

## 2.2 KMP算法中失效函数数组的进一步优化

针对前面定义的失效函数数组，笔者认为，从算法中以空间换时间的角度来看，对于每一次目标串和模式串失配的情况并没有有效利用起来。事实上，如果把模式串右滑后再与目标串匹配的该字符，若与上一个失配的字母相同，那么这次右滑则只是一个跳板。如果在预处理得到失效函数数组的过程中将这块跳板省略，将大大提高在多相同连续字母下的匹配情况。

如表1所示，给出目标串Target和模式串Pattern，当t[3]!=p[3]时，根据原有的失效函数数组，将p[fail[3-1]+1]即p[2]匹配。由于p[3]==p[2]==p[1]==p[0]==’a’，显然每次使用相同的字符’a’与t[3]比较毫无意义，必然是失配。由此可总结为，当出现p[j]==p[fail[j-1]+1]时，就会出现上述的无意义滑动情况。因此，当出现该情况时，需要递归将fail[j]=fail[fail[j-1]+1]]，直到p[j]!=p[fail[j-1]+1]或fail[j]==-1为止，此时记录fail[j]为failval[j]。此时匹配算法不变。

表1：失效函数改进前后数组变化对比

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Target[i] | a | a | a | b | a | a | a | a | b |
| Pattern[j] | a | a | a | a | b |  |  |  |  |
| j | 0 | 1 | 2 | 3 | 4 |  |  |  |  |
| fail[j] | -1 | 0 | 1 | 2 | -1 |  |  |  |  |
| failval[j] | -1 | -1 | -1 | -1 | -1 |  |  |  |  |

## 2.3KMP优化算法测试结果及算法分析结论

根据KMP算法本身目标串和主串之间存在许多“部分匹配”（即每趟比较若干字符后才发现不匹配）的情况才显示其优越性的特性，笔者构造了基本以连续重复字母串作为模式串的测试数据组，以让失效函数数组和优化后的失效函数数组达到有所区别的目的。表2表3分别给出了两个数据组测试后的结果。

表2：数据组1

|  |  |  |
| --- | --- | --- |
| 模式串p[]=”abaabc”  目标串t[] | KMP优化前的单字符比较次数 | KMP优化后的单字符比较次数 |
| ababababababababa | 24 | 21 |
| babababababababababaaaaaaaaaaabaabcv | 53 | 48 |
| ababacaababaabaabc | 24 | 23 |

表3：数据组2

|  |  |  |
| --- | --- | --- |
| 模式串p[]=”aaaab”  目标串t[] | KMP优化前的的单字符比较次数 | KMP优化后的单字符的比较次数 |
| aaabaaaab | 12 | 10 |
| aaacaaabaacbaaabaaaab | 34 | 27 |
| ababaaababaa | 21 | 18 |

由此可见，在优化目的达到，使failval[]优化失效函数数组与fail[]原失效函数数组有所区别的时候，优化后的KMP算法都比未优化时效率高。因此对该优化算法的分析结论为，当模式串大部分由重复字符组成时，使用上述优化算法对失效函数数组进行进一步处理将大大提高算法效率。

# 3对比算法：字符串模式匹配的BM算法的代码实现及算法对比测试

## 3.1 BM算法概述：

1977年，Boyer和Moore针对字符串模式匹配提出了一个全新的算法——BM(Boyer-Moore) 算法。匹配过程中，模式从左向右移动，但字符的比较却从右向左进行。BM算法利用两个辅助数组bad\_table[i]，good\_table[i],定义了两个规则：（1）坏字符规则：当文本串中的某个字符跟模式串的某个字符不匹配时，我们称文本串中的这个失配字符为坏字符，此时模式串需要向右移动，移动的位数 = 与坏字符不匹配的字符在模式串中的位置-坏字符在模式串中最右出现的位置。此外，如果"坏字符"不包含在模式串之中，则最右出现位置为-1。即将坏字符在模式串中最右出现的位置与此时失配的目标串中的字符的位置配对。（2）好后缀规则：当字符失配时，后移位数 = 好后缀在模式串中的位置-好后缀在模式串上一次出现的位置，且如果好后缀在模式串中没有再次出现，则为-1。即，让好后缀在模式串中除了尾部之外最右出现的位置与此时失配的目标串中的字符的位置配对。

如表4所示举例，给出目标串Target和模式串Pattern。为简化问题处理，规定字符集为只有26个小写字母的集合。

表4：BM算法概述举例目标串和模式串

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Target[i] | h | e | r | i | s | a | s | i | m | p |
| Pattern[j] | e | x | a | p | l | e |  |  |  |  |
| Target[i] | l | e | e | x | a | m | p | l | e |  |
| Pattern[j] |  |  |  |  |  |  |  |  |  |  |

其坏字符数组如表5所示。其中i值为对应小写字母的ASCII值减去‘a’的ASCII码值。

表5：举例模式串的坏字符数组

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0a | 1b | 2c | 3d | 4e | 5f | 6g |
| bad\_table[i] | 2 | -1 | -1 | -1 | 6 | -1 | -1 |
| i | 7h | 8i | 9j | 10k | 11l | 12m | 13n |
| bad\_table[i] | -1 | -1 | -1 | -1 | 5 | 3 | -1 |
| i | 14o | 15p | 16q | 17r | 18s | 19t |  |
| bad\_table[i] | -1 | 4 | -1 | -1 | -1 | -1 |  |
| i | 20u | 21v | 22w | 23x | 24y | 25z |  |
| bad\_table[i] | -1 | -1 | -1 | 1 | -1 | -1 |  |

预处理模式串得到坏字符数组的代码也比较简单，复杂度为O(p.length())。

void build\_bad\_table(int bad\_table[], string p){

int l = p.length();

for (int i = 0; i < 26; i++) {

bad\_table[i] = -1;

}

for (int i = 0; i < l; i++) {

bad\_table[p[i] - 'a'] = i;

}

}

其好后缀数组如表6所示。好后缀数组good\_table[i]的满足条件如下：模式串P的长度为l=p.length(),存在0≤n<i，p[n]p[n+1]...p[n+l-i-1]==p[i]p[i+1]...p[l-1]，则good\_table[i]=n；如果不存在这样的n，good\_table[i]= -1。

表6：举例模式串的好后缀数组

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0e | 1x | 2a | 3m | 4p | 5l | 6e |
| good\_table[i] | -1 | -1 | -1 | -1 | -1 | -1 | 0 |

预处理得到好后缀数组的代码如下，复杂度最坏情况下近似于O(p.length()^2)。

int goodSuffix(int pl,const string& p){

int goodx = -1;

for (int i = 0; i < pl; i++) {

int ii = i;

int j = pl;

for (; j < p.length(); ii++,j++) {

if (p[ii] != p[j]) {

break;

}

}

if (j == p.length()) {

goodx = i;

}

}

return goodx;

}

void build\_good\_table(int good\_table[], const string& p){

for (int i = 0; i < p.length(); i++) {

good\_table[i] = goodSuffix(i, p);

}

}

在模式串匹配的过程中，每一次移动后，开始自右向左的匹配。当字符匹配失败时，在此时的bad\_table数组和good\_table数组中选择较小值，也就是能使模式串向右滑动更多的方式。可以发现，好后缀规则的存在也避免了只利用坏字符规则时可能使模式串向左滑动的缺陷。

## 3.2BM算法与KMP算法的对比测试结果及算法分析结论

为了对两种算法的性能、效率做具体的评测，笔者随机指定了一段目标串和模式串进行模式匹配，记录一次匹配模式中的右移次数以及10万次匹配时消耗的时间。

目标串Target=“ofasoliditycontractthatcantransferfundstoaspecifiedaddressataspecifiedtimeanddateinthisupdatedversionofthecontracthereisasimpleexample”。

模式串Pattern=”hereisasimpleexample”。

测试结果见表7。

表7：KMP与BM算法的效率对比测评

|  |  |  |
| --- | --- | --- |
| 算法 | KMP | BM |
| 一次匹配中模式串右移的次数 | 116 | 21 |
| 进行10万次匹配消耗的时间（ms） | 160 | 72 |

由以上测试结果，可以得到BM 算法在模式匹配过程中，模式右移次数少，与KMP算法相比效率较高，是一种较为理想的、快速的模式匹配算法。这其实并不难理解，因为BM算法中对于好后缀good\_table[]的思想与KMP算法中求解失效函数中寻找相同前后缀的思想有异曲同工之妙，同时再借助bad\_table[]，效率高于KMP理所应当。

当然，对于要做的准备工作也多了些步骤，但其复杂度在一般情况下是可以接受的，实际近似于O(p.length())。可以肯定的是，当字符匹配的长度越大且匹配的目标文本越随机时，BM算法优势明显。

# 4提出算法：“首字母缩写”问题的字符串模糊匹配算法

## 4.1“首字母缩写”问题概述

字符串匹配问题中的模糊匹配在实际应用中也有着广泛的用途。尤其是对于信息检索问题，用户给出关键词，以高效率搜索给出用户可能需要的信息是一个优秀的搜索引擎不可或缺的。而要提高搜索效率的关键即是提高模糊匹配字符串的算法效率。

其中，笔者将目标串中是否存在包含按照模式串字符排列顺序的字符元素的子串简称为“首字母缩写”问题。比如给出缩写ccpc,，则chinacollegeticprogrammi ngcontest，childcareprovidercertificationsystem等都是符合条件的目标串。进一步地，在一段文本里找到长度最短的“首字母缩写”问题目标串是更符合实际应用的（因为在长文本里找到按顺序排列的该段字符并不困难，而最短的目标串具有搜索意义的可能性最大）。

如表所示，给出目标串Target和模式串Pattern，希望求得目标串中符合“首字母问题”的最短子串Result。分析问题，使用朴素匹配。开始，对于首字母的匹配，i=0，j=0，t[i]==p[j]==’a’，由于目的为给出长度最短的子串，在匹配成功时i++，但j仍保持0。之后每次匹配同时检测t[i]==p[j]和t[i]==p[j+1]，前者匹配成功则要更新result子串的开始位置为此时的i，即选择重复连续字符p[0]中的最后一个。如果出现了t[i]==p[j+1]，则使j++，此后i和j同步更新，因为中间若有连续重复字符，其选择位置靠前的或位置靠后的将不会对result串的长度再产生影响。此外，不难想到要在向前推进i的过程中遇到t[i]==p[0]时使用nextbegin[]数组记录位置，nextbegin的记录也可以做一个简单优化处理，即选择重复连续字符中的最后一个。当上一轮匹配结束后，取出nextbegin数组中的下一个元素位置重复上述匹配。上述匹配的复杂度为O(t.length()^2)。该算法效率是非常低下的。

表8：“首字母缩写”问题举例目标串、模式串

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Target[i] | s | h | a | n | g | h | a | i | c | p | c |
| Pattern[j] | a | c |  |  |  |  |  |  |  |  |  |

## 4.2提出算法：对目标串进行预处理的NextPos算法

下面给出一种对此类问题的处理算法。与常见的精确字符串匹配对模式串进行预处理不同，处理该问题，笔者提出对目标串进行预处理。建立二维数组nextpos[i][j]，记录字符j在目标串T[i]后的下一个最近位置。为简化问题处理，规定字符集为只有26个小写字母的集合。j为相应字母的ASCII码值减去a的ASCII码值。如果该字母不存在，则将nextpos[i][j]值置为目标字符串的长度，即目标字符串下标达不到的位置。由表给出的目标串T为例，其nextpos[i][j]数组如表9所示。其中，\*表示目标串下标取不到的值，统一

规定为目标串的长度，这里取11。

计算nextpos数组的算法如下。通常情况下，此预处理的算法复杂度为O(t.length()\*num(字符集元素数))。

void NextPosArray(String t,int nextpos[][26]){

int pos=new int[26];

for(int i=0;i<26;i++){

pos[i]=t.length();.

}

for(int i=t.length()-1;i>=0;i--){

for(int j=0;j<26;j++){

nextpos[i][j]=pos[j];

}

pos[t[i]-’a]=i;

}

delete []pos;

}

表9：举例目标串nextpos数组

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j\P[i] | 0s | 1h | 2a | 3n | 4g | 5h | 6a | 7i | 8c | 9p | 10c |
| 0a | 2 | 2 | 6 | 6 | 6 | 6 | \* | \* | \* | \* | \* |
| 1b | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 2c | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 10 | 10 | \* |
| 3d | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 4e | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 5f | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 6g | 4 | 4 | 4 | 4 | \* | \* | \* | \* | \* | \* | \* |
| 7h | 1 | 5 | 5 | 5 | 5 | \* | \* | \* | \* | \* | \* |
| 8i | 7 | 7 | 7 | 7 | 7 | 7 | 7 | \* | \* | \* | \* |
| 9j | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 10k | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 11l | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 12m | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 13n | 3 | 3 | 3 | \* | \* | \* | \* | \* | \* | \* | \* |
| 14o | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 15p | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | \* | \* |
| 16q | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| …… | …… | | | | | | | | | | |
| 25z | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |

此后的模糊匹配中，从每一个t[i]=p[0]开始的每一次匹配只需经过模式串p长度次跳跃，如果此时跳跃到的位置为t.length()，则整个模糊匹配终止。目标串T中有n个字符为p[0]，则整个算法的复杂度为O（n\*p.length()），综合考虑算法效率远远大于朴素匹配。匹配的算法如下。

void NextPos(String t,String p，int nextpos[][26]){

int begin,ansbegin,ansend;

int tLength=t.length();

for(begin=nextpos[0][p[0-’a’];begin<tLength;begin=nextpos[begin][p[0]-’a’]){

int nowpos=begin;

bool flag=1;

for(int j=1;j<p.length();j++) {

nowpos=nextpos[nowpos][t[j]-’a];

if(nowpos>tLength){

flag=0;

break;

}

}

if(flag==0){

return;

}

if(nowpos-begin<ansend-ansbegin){

ansend=nowpos;

ansbegin=begin;

}

}

}

## 4.3NextPos算法测试结果及算法分析结论

综合考虑NextPos数组储存算法本身在处理中间有大量间隔字符的优越性和“首字母问题”本身在实际问题中的应用，构造测试数据及其余与朴素模式匹配的最小执行单元数如表10所示。

表10：数据组3

|  |  |  |  |
| --- | --- | --- | --- |
| 目标串 | 模式串 | 朴素算法匹配次数 | NextPos算法跳跃次数 |
| aadead | ae | 5 | 3 |
| shanghaicpc | ac | 4 | 2 |
| aaabbbaa  abbbccc | abc | 21 | 12 |
| howdeliciou  sandfreshitis | oishii | 19 | 8 |
| chinacollegi  ateprogrammingcontest | ccpc | 10 | 6 |
| heqihaosunyifei | hqhsyf | 12 | 6 |
| woyidingne  ngshangqinghuaduaxue | qhdx | 7 | 3 |

由此可见，当对目标串进行预处理后得到nextpos数组后进行的模糊匹配的效率甚至高达朴素模式匹配的两倍，与此同时，运用先对目标串进行预处理的思想还可以适用于一目标串和多模式串的模糊匹配问题。由此还应得到突破思维局限性的经验，对于模式串进行预处理可以有效达到比较过程中无需回溯，提高效率的目标，而对目标串进行预处理同样能达到目的，并且在解决模糊匹配时对目标串进行预处理显然更为合理。

美中不足的是，因为运用了以空间换时间的理念，对于一些空间的利用是浪费多余的，比如对于模式串中并没有出现的字符，也需要在对目标串的预处理中占用空间去记录。

5优化算法:多模式串匹配中的字典树和DP算法优化  
5.1字典树概述

当需要多个模式串去匹配一个目标串时，使用单模式串的匹配算法对每一个模式串与目标串进行匹配，在模式串个数和长度量较大时的时间复杂度显然难以接受，因为对于前缀有所重复的模式串来说，再次进行匹配显然是不必要的。为了避免这种无效匹配，在多模式串的匹配中，笔者使用字典树的数据结构进行算法的构建。

字典树，Trie树，也叫前缀树，它是一个树形的结构。是一种专门用来处理字符串匹配的数据结构，用来解决在一组字符串中快速查找某个字符串的问题。谷歌，百度这种搜索引擎，输入框的关键词提示功能，底层原理就是使用了这种数据结构。

Tire树是一种有序树，用于保存关联数组，一个结点的所有的子孙都有共同的前缀。Trie的本质就是利用字符串的公共前缀，把重复的前缀合并在一起。比如对“hello”，“hi”，“hey”,“her”，“hello”，“seek”，“see”，进行字典树的构建，字典树如图1所示。

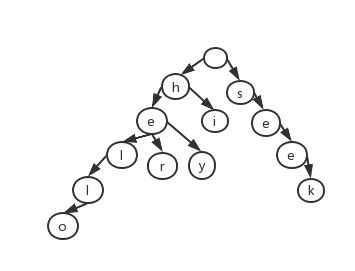
可以看出字典树有以下三个特性：（1）根节点是空的，除了根结点之外，每个结点都只包含一个字符；（2）从根结点到某一个结点的路径上的字符连起来，就是该结点对应的字符串；（3）每个结点对应的子结点包含的字符串不同。由此,这种数据结构能够有效处理上文提到的重复无效匹配前缀的问题。

图1：举例构建的字典树数据结构

对于多叉树的储存，传统的做法是借助散列表的思想，通过一个下标与字符串一一映射的数组，来存储结点的指针。为了简化问题处理，我们规定字符集为26个小写字母。我们使用一个长度为26的数组children[]，数组的下标即为对应字母的ASCI码值减去’a’的ASCII码值，数组中储存的为对应字母子节点的指针，如果某个字符的子节点不存在，就在对应下标位置存储NULL。同时给每个字符结点增加一个标记tail，来记录此时该字符是否是一个字符串的结尾。在Trie树中插入一个字符串的代码如下。

void Trie::insert(const string& p){  
 TrieNode\* p = root;

for (int i = 0; i < p.length(); ++i) {

int index = text[i] - 'a';

if (p.children[index] ==NULL) {

TrieNode\* newNode = new TrieNode(text[i]);

p.children[index] = newNode;

}

p = p.children[index];

}

p->tail=1;

}

在字典树中查找一个字符串的算法也非常简单，代码如下。

bool find(const string& t){

TrieNode\* p = root;

for (int i = 0; i < t.length; ++i) {

int index = pattern[i] - 'a';

if (p.children[index] == NULL) {

return false;

}

p = p.children[index];

}

if(p->tail=0){

return false;

}

return true;

}

每次查询的时候，如果要查询的字符串长度是k,那么我们只需要对比大约k个节点就能完成查询操作，跟原本的那组字符串的长度和没有任何关系，效率非常高。

## 5.2 DP思想对多模式串的匹配的算法优化

给出一个以多个模式串为元素的字符串集和一个目标串，判断目标串是否由模式串集中的模式串交替拼接而成，如果是，给出最少由几个字符串拼接而成，是一个多模式串匹配中的经典问题。此时字典树形式的构造和DP（Dynamic Programming）的思想能够有效结合起来，高效率地解决问题。

算法思想概述为：以dp[i]表示询问串的前i类，结尾的串属于多模式串集的最短步数。O(n)的时间枚举转移，每次枚举一个比i小的j，当j+1到i的子串在模式串集中时，dp[i]=min(dp[j]+1)，(0≤j<i，枚举所有j)。这是一个较为简单的状态转移方程。实现代码如下。

void dp(int dp[]，const string& t){  
 int len=t.length();

for(int i=0;i<len;i++){

dp[i]=INF;//一个无穷大值

}

string sstr=t.substr(0,1);

if(TrieTree.find(sstr)){  
 dp[0]=1;

}

else{  
 dp[0]=0;

}

for(int i=1;i<l;i++){

for(int j=0;j<i;j++){  
 sstr=t.substr(j,i);

if(TrieTree.find(sstr)){

dp[i]=min(dp[i],dp[j]+1):

}

}

}

}

dp过程进行完毕之后，考察dp[t.length()-1]的值，若为INF（无穷大），则说明目标串并不是全由模式串集中的字符串组成的；若是一个正常值，则代表其中模式串的最小个数。

## 5.3单个模式串依次匹配与DP方法算法测试结果及算法分析结论

为了对DP算法的增进的性能、效率做具体的评测，笔者给出了目标串和模式串集，记录 10万次匹配时消耗的时间。

给出目标串Target=“femaleccpc”,模式串一Pattern1=“female”，模式串一Pattern1=“female”，

模式串二Pattern2=“male”，模式串三Pattern3=“cc”，模式串一Pattern4=“ccpc”，算法测试结果如表11。

表11：一个模式串依次匹配与DP算法的效率测评

|  |  |  |
| --- | --- | --- |
| 算法 | 依次匹配 | DP |
| 进行10万次匹配消耗的时间（ms） | 101 | 36 |

由此可见，利用字典树的基础之上，在多模式串匹配的问题上合理运用DP思想解决问题，能够进一步提高算法的效率，且思想简单成体系，便于理解，能够高效解决问题。

**6 结 语**

对于字符串匹配问题，基于不同的算法思想

可以建立不同的预处理模式，对于预处理结果采用不同形式的存储结构。

对于KMP和BM算法采用了对模式串进行预处理的形式，用较为简单的数组形式存储结果；对于模糊匹配，则可以采用对目标串进行NP预处理的形式，开拓了思路且算法简单；在多模式串匹配的过程中，运用了字典树Trie的数据结构，合理运用动态规划DP的思想解决问题，进一步优化算法。

且通过实验证实，每一次对算法的改进和对数据结构的合理运用都能够得到较为理想的结果。

**参考文献**

[1]巫喜红,凌捷.BM模式匹配算法剖析[J].计算机工程与设计,2007(01):29-31.DOI:10.16208/j.issn1000-7024.2007.01.010.

[2]欧嵬,吴纯青.几种字符串匹配算法的分析和比较[J].微处理机,2007(04):59-61.

[3]孙友仓.多模式匹配算法的性能分析[J].电子设计工程,2010,18(01):17-18.DOI:10.14022/j.cnki.dzsjgc.2010.01.039.

[4]严蔚敏,李冬梅,吴伟民. 数据结构[M].人民邮电出版社:, 201502.28