字符串查找算法

20123211 高雨晴

0. 概述

- 实验内容：实现KMP算法和Sunday算法，尝试结合二者实现一个更加优化的算法。
- 实验方法：使用C++语言，通过MinGW-w64 C/C++编译器
- 实验结果：输出结果和预期相符。

1. 实验设计

1.1 算法实现原理

1.1.1 KMP算法

假设在字符串（text）"ABC ABCDAB ABCDABCDABDE"中搜索（pattern）"ABCDABD"。使用两个循环变量m和i，其中m代表text内匹配pattern的当前查找位置，i表示匹配字符串pattern当前作比较的字符位置。

```
             1         2
m: 01234567890123456789012
T: ABC ABCDAB ABCDABCDABDE
P: ABCDABD
i: 0123456
```

从P和T的开头比起。比对到T[3]（空格）时发现P[3]（D）与之不符。所以下次直接略过T[1]~T[3]，因为已经知道了它们和P[0]也不符合。因此令m=4，i=0.

```
             1         2
m: 01234567890123456789012
T: ABC ABCDAB ABCDABCDABDE
P:     ABCDABD
i:     0123456
```

比较后发现，“ABCDAB”字符串是相符的，但下一个字符又不相同。注意到“AB”在“ABCDAB”的头尾均有出现，则尾端的“AB”可以作为下次比较的起始点。因此令m=8，i=2.

```
             1         2
m: 01234567890123456789012
T: ABC ABCDAB ABCDABCDABDE
P:         ABCDABD
i:         0123456
```

在m=10的地方又出现了不相符。类似的，令m=11，i=0.

```
            1         2
m: 012345678901234567890 12
T: ABC ABCDAB ABCDABCDABDE
P:             ABCDABD
i:             0123456
```

这时T[17]和P[6]不相同，但出现了和之前一样的情况。令m=15，i=2.

```
            1         2
m: 012345678901234567890 12
T: ABC ABCDAB ABCDABCDABDE
P:                 ABCDABD
i:                 0123456
```

此时找到了完全匹配的字符串，起始位置位于T[15]。

1.1.2 Sunday算法

仍然用上面的例子。

```
            1         2
m: 012345678901234567890 12
T: ABC ABCDAB ABCDABCDABDE
P: ABCDABD
i: 0123456
```

P[3]不与T[3]匹配，但T[7]='D'存在于P内，于是使T[7]和P[3]对齐。

```
            1         2
m: 012345678901234567890 12
T: ABC ABCDAB ABCDABCDABDE
P:     ABCDABD
i:     0123456
```

此时T[10]又不匹配，但T[11]='A'存在于P中，P向右移动到P[0]和T[11]对齐。

```
            1         2
m: 012345678901234567890 12
T: ABC ABCDAB ABCDABCDABDE
P:             ABCDABD
i:             0123456
```

T[17]不匹配，和上面一样，P向右移动至P[3]和T[18]对齐。

```
            1         2
m: 012345678901234567890 12
T: ABC ABCDAB ABCDABCDABDE
P:                 ABCDABD
i:                 0123456
```

完成匹配，Pattern在text中的起始位置是T[15]。

1.2 算法设计与分析

1. 以LSP数组作为预处理数组

```
void lps_func(string txt, vector<int>&Lps){// Lps[i]存储的是模式串前i+1个字符所组成的子串中，最大的相同前后缀。
    Lps[0] = 0;
    int len = 0;
    int i=1;
    while (i<txt.length()){
        if(txt[i]==txt[len]){
            len++;
            Lps[i] = len;
            i++;
            continue;
        }
        else{
            if(len==0){
                Lps[i] = 0;
                i++;
                continue;
            }
            else{
                len = Lps[len-1];
                continue;
            }
        }
    }
}
```

2. KMP算法

```
void KMP(string pattern,string text){
    int n = text.length();
    int m = pattern.length();
    vector<int>Lps(m);

    lps_func(pattern,Lps); // 构造lsp数组

    int i=0,j=0;
    while(i<n){
        if(pattern[j]==text[i]){i++;j++;} // 如果匹配，则继续

        if (j == m) {
            cout<<i - m <<' ';     // 如果j==m，则找到了，输出下标
                                   // 并更新j为最后匹配的字符的Lps
            j = Lps[j - 1];
        }
        else if (i < n && pattern[j] != text[i]) {  // 匹配失败的情况
            if (j == 0)           // 如果j变成0，则下标i加1
                i++;
            else
                j = Lps[j - 1];  // 更新j为最后匹配的字符的Lps
```

```
        }
    }
}
```

循环最多进行2n次（n为待匹配字符串的长度），故时间复杂度为O(n)。

3. Sunday算法

```
int SundaySearch(string text,string pattern){
    int sn=pattern.size();
    int tn=text.size();
    if(sn<=0||tn<=0)
        return -1;
    int i=0,j=0,k;
    int m=tn;
    for(;i<sn;){
        if(pattern[i]!=text[j]){
            for(k=tn-1;k>=0;k--){
                if(text[k]==pattern[m])
                    break;
            }
            i=m-k;
            j=0;
            m=i+tn;
        }
        else{
            if(j==tn-1)
                return i-j;
            i++;
            j++;
        }
    }
}
```

平均时间复杂度为O(n)，当需要匹配的两组字符串中重复字符过多时，最差情况的时间复杂度为O(n∗m)。

2. 实验结果说明与分析

2.1 实验设置

以上面的例子作为测试例。

```
int main()
{
    string text = "ABC ABCDAB ABCDABCDABDE";
    string pattern = "ABCDABD";
    KMP(pattern, text);
    cout << SundaySearch(pattern, text) << endl;

    system("pause");
    return 0;
}
```

2.2 实验结果以及相应的分析

输出结果：

```
15 15
```

即两种算法得出的结果均为Pattern在Text中的起始位置是T[15]. 这与我们自行计算的结果相符。

3. 改进

KMP算法的核心思想是：一个词在不匹配时本身就包含足够的信息来确定下一个匹配可能的开始位置，利用这一特性以避免重新检查先前配对的字符。

Sunday算法的核心思想是：在匹配过程中，模式串并不被要求一定要按从左向右进行比较还是从右向左进行比较，它在发现不匹配时，算法能跳过尽可能多的字符以进行下一步的匹配，从而提高了匹配效率。

然而Pattern的首字符重复越多，Sunday算法的优越性越不明显。例如在"baaaabaaaabaaaabaaaa"中查找"aaaaa"，使用sunday算法和暴力查找的效率是一样的。因此有必要将Sunday算法和KMP算法相结合进行改进。一个可行的方法是，在开始匹配前检验模式串的首个字符是否重复，如果没有重复则直接按照Sunday算法进行匹配；如果重复，将重复字符压缩为一个字符，避免了无效的字符匹配。如果压缩串匹配成功，则回溯判断压缩串前面的字符是否是首字符。

```
if(pattern[0]==pattern[1]){
    int count = 0;
    while(pattern[count]==pattern[count+1]){
        count ++;// 判断重复次数
    }
    for(int j=0; count<strlen(pattern);count++,j++){
        te[j]=pattern[count];// 将pattern压缩
    }
    int first = SundaySearch(text,te); // 返回匹配text的首个元素的位置
    int temp = strlen(pattern)-strlen(te);// 返回首字母重复的位数temp
    while(temp!=0){// 循环temp次，判断匹配串前面的字符
        if(text[first--]!= pattern[0]){// 若发现和首字符不同则返回-1
            return -1;
            break;
        }
        temp --;
    }
}
```

4. 附录

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

// LPS function
void lps_func(string txt, vector<int>&Lps){
    Lps[0] = 0;
```

```
    int len = 0;
    int i=1;
    while (i<txt.length()){
        if(txt[i]==txt[len]){
            len++;
            Lps[i] = len;
            i++;
            continue;
        }
        else{
            if(len==0){
                Lps[i] = 0;
                i++;
                continue;
            }
            else{
                len = Lps[len-1];
                continue;
            }
        }
    }
}

// KMP Functions
void KMP(string pattern,string text){
    int n = text.length();
    int m = pattern.length();
    vector<int>Lps(m);

    lps_func(pattern,Lps);

    int i=0,j=0;
    while(i<n){
        if(pattern[j]==text[i]){i++;j++;}

        if (j == m) {
            cout<<i - m <<' ';
            j = Lps[j - 1];
        }
        else if (i < n && pattern[j] != text[i]) {
            if (j == 0)
                i++;
            else
                j = Lps[j - 1];
        }
    }
}

// sunday search
int SundaySearch(string text,string pattern){
    int sn=pattern.size();
    int tn=text.size();
    if(sn<=0||tn<=0)
        return -1;
    int i=0,j=0,k;
```

```
    int m=tn;
    for(;i<sn;){
        if(pattern[i]!=text[j]){
            for(k=tn-1;k>=0;k--){
                if(text[k]==pattern[m])
                    break;
            }
            i=m-k;
            j=0;
            m=i+tn;
        }
        else{
            if(j==tn-1)
                return i-j;
            i++;
            j++;
        }
    }
}

int main()
{
    string text = "ABC ABCDAB ABCDABCDABDE";
    string pattern = "ABCDABD";
    KMP(pattern, text);
    cout << SundaySearch(pattern, text) << endl;

    system("pause");
    return 0;
}
```