

一种字符串匹配算法分享

—————KMP算法



背景介绍

- 在编辑文本过程中，我们经常需要在文本中找到某个模式的所有出现位置。比如，一段正在被编辑的文本构成一个文件，而所要搜寻的模式是用户正在输入的特定的关键字。有效解决这个问题算法称为字符串匹配算法，这种算法能够极大提高编辑文本程序时的响应效率。

暴力匹配算法

- 假如有一个长度为 n 的字符串数组 $S[0\dots n-1]$,有一个长度为 m 的模式串数组 $P[0\dots m-1]$,其中 $m \leq n$,且 S 和 P 的元素来自一个有限字符集,例如 $\{1,2,3\}$ 、 $\{a,b,c,\dots,z\}$ 等。查找数组 P 在数组 S 中出现的位置,如果存在 s , $0 \leq s \leq n-m-1$,且
- $S[s..s+m-1]=P[0..m]$ 即 $S[s+j]=P[j]$, $1 \leq j \leq m$, 则模式串 P 在字符串 S 中以偏移 s 的长度出现, 现在需要求出 s 的值。
- 通过暴力匹配方式求解 s 的值

暴力匹配算法

- 假设现在字符串S匹配到i位置，模式串P匹配到j位置，则有如下的基本匹配思想：
- 如果当前字符匹配成功($S[i]==P[j]$)则 $i++$, $j++$,继续匹配下个字符
- 如果失配($S[i]!=P[j]$),令 $i=i-(j-1)$, $j=0$ 。失配后i回溯，j值为零。

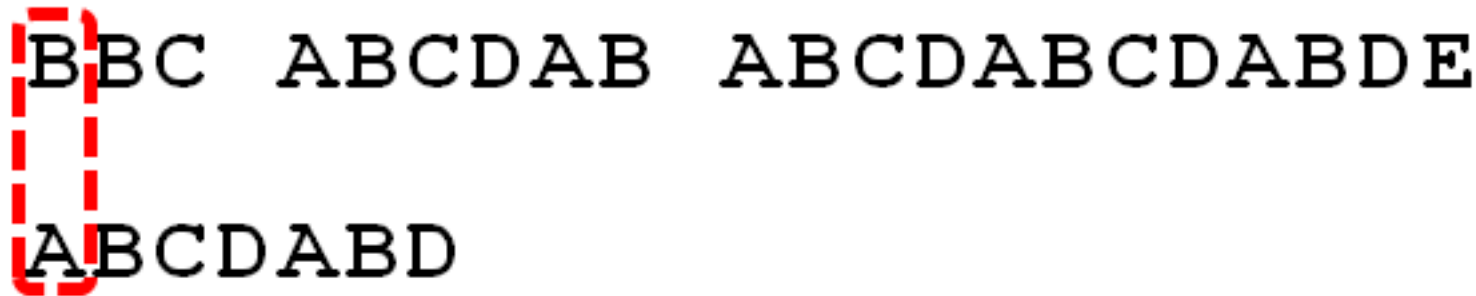
暴力匹配算法

```
1 int ViolentMatch(char* s, char* p)
2 {
3     int sLen = strlen(s);
4     int pLen = strlen(p);
5
6     int i = 0;
7     int j = 0;
8     while (i < sLen && j < pLen)
9     {
10         if (s[i] == p[j])
11         {
12             // 如果当前字符匹配成功 (s[i] == p[j]) , 则 i++, j++
13             i++;
14             j++;
15         }
16         else
17         {
18             // 如果匹配失败 (s[i] != p[j]) , 令 i = i - (j - 1), j = 0
19             i = i - j + 1;
20             j = 0;
21         }
22     }
23     // 匹配成功, 返回模式串p在文本串s中的位置, 否则返回-1
24     if (j == pLen)
25         return i - j;
26     else
27         return -1;
28 }
```

暴力匹配算法

举个例子，如果给定文本串S“BBC ABCDAB ABCDABCDABDE”，和模式串P“ABCDABD”，现在要拿模式串P去跟文本串S匹配，整个过程如下所示：

S[0]为B，P[0]为A，不匹配，令 $i = i - (j - 1)$ ， $j = 0$ ，S[1]跟P[0]匹配，相当于模式串要往右移动一位（ $i=1$ ， $j=0$ ）



BBC ABCDAB ABCDABCDABDE
ABCDABD


暴力匹配算法

BBC ABCDAB ABCDABCDABDE
ABCDABD


BBC ABCDAB ABCDABCDABDE
ABCDABD

暴力匹配算法

BBC ABCDAB ABCDABCDABDE
ABCDABD



BBC ABCDAB ABCDABCDABDE
ABCDABD



暴力匹配算法

而 $S[5]$ 肯定跟 $P[0]$ 失配。因为在之前第4步匹配中，我们已经得知 $S[5] = P[1] = B$ ，而 $P[0] = A$ ，即 $P[1] \neq P[0]$ ，故 $S[5]$ 必定不等于 $P[0]$ ，所以回溯过去必然会导致失配。那有没有一种算法，让 i 不往回退，只需要移动 j 即可呢？

这种算法就是KMP算法，它利用之前已经部分匹配这个有效信息，保持 i 不回溯，通过修改 j 的位置，让模式串尽量地移动到有效的位置。

Knuth-Morris-Pratt 字符串查找算法，简称为“KMP算法”，常用于在一个文本串 S 内查找一个模式串 P 的出现位置，这个算法由Donald Knuth、Vaughan Pratt、James H. Morris三人于1977年联合发表，故取这3人的姓氏命名此算法。

KMP算法

假设现在文本串S匹配到 i 位置，模式串P匹配到 j 位置，则匹配思想：

如果 $j = -1$ ，或者当前字符匹配成功（即 $S[i] == P[j]$ ），都令 $i++$ ， $j++$ ，继续匹配下一个字符；

如果 $j \neq -1$ ，且当前字符匹配失败（即 $S[i] \neq P[j]$ ），则令 i 不变， $j = \text{next}[j]$ 。此举意味着失配时，模式串P相对于文本串S向右移动了 $j - \text{next}[j]$ 位。

KMP算法

```
1 int KmpSearch(char* s, char* p)
2 {
3     int i = 0;
4     int j = 0;
5     int sLen = strlen(s);
6     int pLen = strlen(p);
7     while (i < sLen && j < pLen)
8     {
9         //如果j == -1, 或者当前字符匹配成功 (s[i] == p[j]), 则令i++, j++
10         if (j == -1 || s[i] == p[j])
11         {
12             i++;
13             j++;
14         }
15         else
16         {
17             //如果j != -1, 且当前字符匹配失败 (s[i] != p[j]), 则令 i 不变, j = next[
18             //next[j] 即为j对应的next值
19             j = next[j];
20         }
21     }
22     if (j == pLen)
23         return i - j;
24     else
25         return -1;
26 }
```

KMP算法

BBC ABCDAB ABCDABCDABDE
ABCDABD

BBC ABCDAB ABCDABCDABDE
ABCDABD

当 $S[10]$ 跟 $P[6]$ 匹配失败时，KMP不是跟暴力匹配那样简单的把模式串右移一位，而是：“如果 $j \neq -1$ ，且当前字符匹配失败（即 $S[i] \neq P[j]$ ），则令 i 不变， $j = \text{next}[j]$ ”，即 j 从6变到2，所以相当于模式串向右移动的位数为 $j - \text{next}[j]$ （ $j - \text{next}[j] = 6 - 2 = 4$ ）。

KMP算法

- next数组是什么？
- 如何求解next数组？
- next[j]表示在模式串P的序号为j的字符前面字符串中存在一个最大值t，满足 $P_0P_1..P_{t-1} = P_{j-t}P_{j-t+1}..P_{j-1}$ ，则 $next[j]=t; 0 \leq t \leq j-1$
- 其实next 数组就是当前字符前的子字符串中最长相同前缀后缀的长度。

KMP算法

如果给定的模式串为“abab”，那么它的各个子串的前缀后缀的公共元素的最大长度如下表格所示：

模式串	a	b	a	b
最大前缀后缀公共元素长度	0	0	1	2

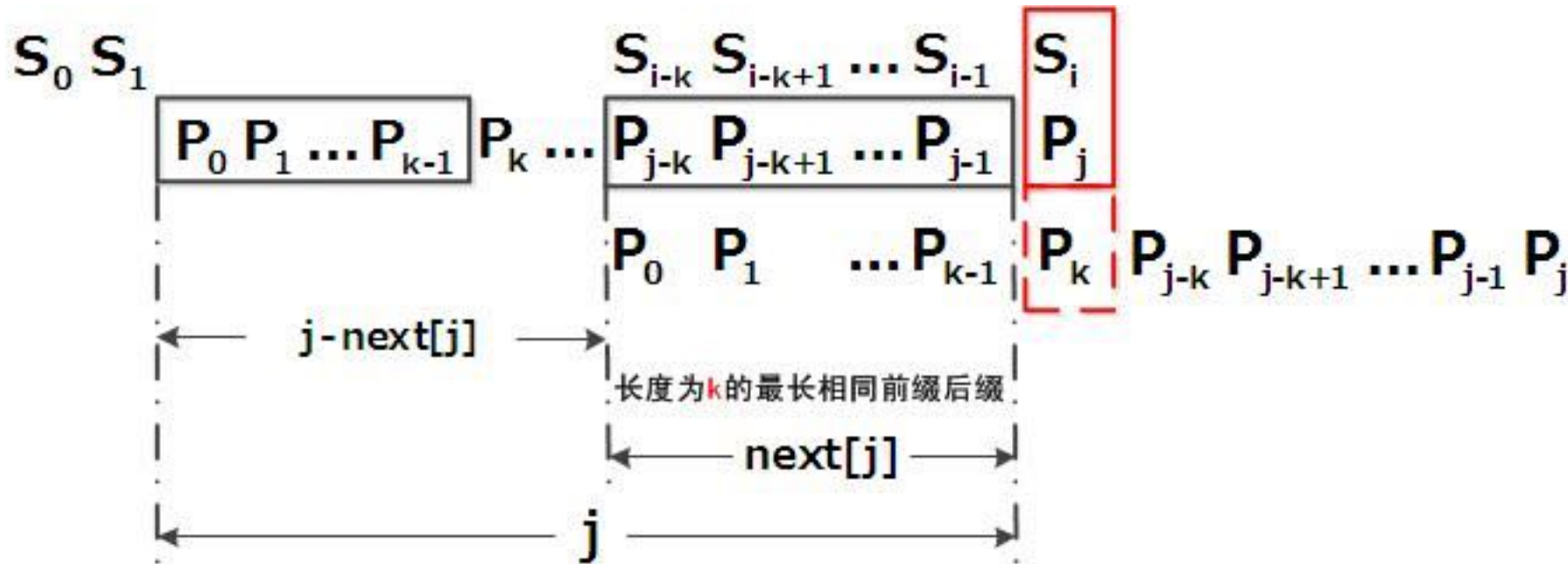
比如对于字符串aba来说，它有长度为1的相同前缀后缀a；而对于字符串abab来说，它有长度为2的相同前缀后缀ab（相同前缀后缀的长度为 $k + 1$ ， $k + 1 = 2$ ）。

模式串	a	b	a	b
next数组	-1	0	0	1

比如对于aba来说，第3个字符a之前的字符串ab中有长度为0的相同前缀后缀，所以第3个字符a对应的next值为0；而对于abab来说，第4个字符b之前的字符串aba中有长度为1的相同前缀后缀a，所以第4个字符b对应的next值为1（相同前缀后缀的长度为 k ， $k = 1$ ）。

KMP算法

匹配失败, $j = \text{next}[j]$, 模式串向右移动的位数为: $j - \text{next}[j]$ 。换言之, 当模式串的后缀 $p_{j-k} p_{j-k+1}, \dots, p_{j-1}$ 跟文本串 $s_{i-k} s_{i-k+1}, \dots, s_{i-1}$ 匹配成功, 但 p_j 跟 s_i 匹配失败时, 因为 $\text{next}[j] = k$, 相当于在**不包含** p_j 的模式串中有最大长度为 k 的相同前缀后缀, 即 $p_0 p_1 \dots p_{k-1} = p_{j-k} p_{j-k+1} \dots p_{j-1}$, 故令 $j = \text{next}[j]$, 从而让模式串右移 $j - \text{next}[j]$ 位, 使得模式串的前缀 $p_0 p_1, \dots, p_{k-1}$ 对应着文本串 $s_{i-k} s_{i-k+1}, \dots, s_{i-1}$, 而后让 p_k 跟 s_i 继续匹配



KMP算法

模式串的各个子串	前缀	后缀	最大公共元素长度
A	空	空	0
AB	A	B	0
ABC	A,AB	C,BC	0
ABCD	A,AB,ABC	D,CD,BCD	0
ABCD A	A,AB,ABC,ABCD	A,DA,CDA,BCDA	1
ABCDAB	A,AB,ABC,ABCD,ABCD A	B,AB,DAB,CDAB,BCDAB	2
ABCDABD	A,AB,ABC,ABCD,ABCD A ABCDAB	D,BD,ABD,DABD,CDABD BCDABD	0

KMP算法

- 计算next数组的思想
- 已知 $\text{next}[j]=k$, 求 $\text{next}[j+1]$
- 若 $p[k]==p[j]$, 则 $\text{next}[j+1]=\text{next}[j]+1=k+1$;
- 若 $p[k] \neq p[j]$, 如果此时 $p[\text{next}[k]] == p[j]$, 则 $\text{next}[j+1] = \text{next}[k] + 1$, 否则继续递归前缀索引 $k = \text{next}[k]$, 而后重复此过程。

KMP算法

- 可以通过递归求得next数组，代码如下

```
1 void GetNext(char* p,int next[])
2 {
3     int pLen = strlen(p);
4     next[0] = -1;
5     int k = -1;
6     int j = 0;
7     while (j < pLen - 1)
8     {
9         //p[k]表示前缀, p[j]表示后缀
10        if (k == -1 || p[j] == p[k])
11        {
12            ++k;
13            ++j;
14            next[j] = k;
15        }
16        else
17        {
18            k = next[k];
19        }
20    }
21 }
```

KMP算法

- next数组的优化

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
-1	0	0	1

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
-1	0	0	1

问题出在不该出现 $p[j] = p[\text{next}[j]]$ 。理由是：
当 $p[j] \neq s[i]$ 时，下次匹配必然出现 $p[\text{next}[j]]$ 跟 $s[i]$ 不匹配，所以当出现 $p[j] = p[\text{next}[j]]$ 的时候，需要再次递归 $\text{next}[j] = \text{next}[\text{next}[j]]$

KMP算法

模式串	a	b	a	b
最大长度值	0	0	1	2
未优化next数组	$\text{next}[0] = -1$	$\text{next}[1] = 0$	$\text{next}[2] = 0$	$\text{next}[3] = 1$
索引值	p_0	p_1	p_2	p_3
优化理由	初值不变	$p[1] \neq p[\text{next}[1]]$	因 p_j 不能等于 $p[\text{next}[j]]$ ，即 $p[2]$ 不能等于 $p[\text{next}[2]]$	$p[3]$ 不能等于 $p[\text{next}[3]]$
措施	无需处理	无需处理	$\text{next}[2] = \text{next}[\text{next}[2]] = \text{next}[0] = -1$	$\text{next}[3] = \text{next}[\text{next}[3]] = \text{next}[1] = 0$
优化的next数组	-1	0	-1	0

总结

- 暴力匹配算法时间复杂度 $O(m*n)$
- KMP算法时间复杂度 $O(m*n)$

