实现并对比三种基本的字符串匹配算法

摘　要

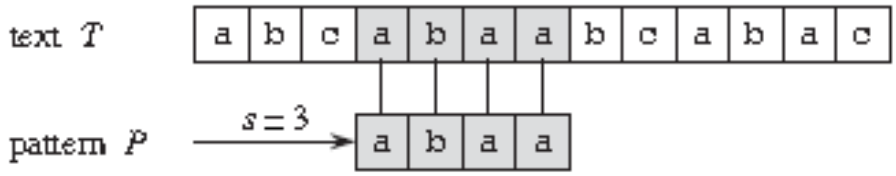
首先对三种基本字符串匹配算法进行了详细分析和说明，再编程实现。创新拓展研究了 Boyer-Moore 算法，进行了分析和编程实现。让四种算法对数据量极大的文本，进行子串的查询处理，并分析算法运行时间效率，并对所有输出的匹配位置结果进行相互对比验证，以证明算法设计和实现的正确性。为了分析不同数据规模对不同算法的影响程度，通过改变文本的数据量大小，用相同的子串进行模式查找，通过对运行时间的比较以获得数据规模对算法的影响，并利用 MATLAB 绘制效率图进一步直观分析。

关键词：字符串匹配 Naive 算法　Rabin-Karp 算法 KMP 算法　Boyer-Moore 算法 时间复杂度 算法效率图

目录

问题描述

文本是一个长度为 n 的数组$T[1..n]$，模式是一个长度为$m \le n$的数组$P[1..m]$，T 和 P 中的元素都属于有限的字母表 Σ 表，如果$0 \le s \le n-m$，并且$T[s+1..s+m]=P[1..m]$，即对 1≤j≤m，有 T[s+j] = P[j]，则称模式 P 完成了和 T 的匹配，且 P 在 T 中出现的位移为 s 并称 s 是一个有效位移。



比如上图中，目标是找出在文本 T = abcabaabcabac 中模式 P = abaa 的所有出现有效位移。从图中易知该模式 P 在此文本 T 中仅出现一次，即在位移 s = 3 处，则位移 s = 3 是一个有效位移。

需求分析

假设和简化

用$\sum^*$表示包含所有有限长度的字符串的集合，该字符串是由字母表$\sum$ 中的字符组成。长度为零的空字符串用∈表示，也属于$\sum^*$。一个字符串 x 的长度用$|x|$来表示。两个字符串 x 和 y 的连结用 xy 表示，长度为$|x|+|y|$，由 x 的字符后接 y 的字符构成。如果对于某个字符串 $y \in \sum^*$ 有 $x = wy$，则称字符串 w 是字符串 x 的前缀，记作 $w \rhd x$。如果 $w \rhd x$，则 $|w| \le |x|$。类似地，如果对于某个字符串 y 有 $x = yw$，则称字符串 w 是字符串 x 的后缀，记作 $w \lhd x$。和前缀类似，如果 $w \lhd x$，则 $|w| \le |x|$。例如，$ab \rhd abcca$ 和 $cca \lhd abcca$。空字符串∈同时是任何一个字符串的前缀和后缀。对于任意字符串 x 和 y 以及任意字符 a，当且仅当 $xa \lhd ya$ 时，我们有 $x \lhd y$。

为了使符号简结，模式$P[1..m]$的由 k 个字符组成的前缀$P[1..k]$记作P_k。因

此，$P_0=\in$，$P_m=P=P[1..m]$。与此类似，文本 T 中由 k 个字符组成的前缀记为T_k。

采用这种记号，我们能够把字符串匹配问题表述为：找到所有偏移$s(0\le s\le n-m)$，使得$P\lhd T_{s+m}$。

功能和运用

在编辑文本程序过程中，经常需要在文本中找到某个模式的所有出现的位置。典型的情况就是，一段正在被编辑的文本构成一个文件，而所要搜索的模式是用户正在输入的特定的关键字，有效的解决这个问题的算法就是字符串匹配算法，该算法能够极大提高编辑文本程序时的响应效率。

在其他很多应用中，字符串匹配算法用于在 DNA 序列中搜索特定的序列。在网络搜索引擎中也需要用这种方法来找到所需要查询的网页地址。

算法和数据结构设计

通过查阅资料整理可得，解决字符串匹配的算法包括：朴素匹配算法（Naive Algorithm）、Rabin-Karp 算法、有限自动机算法（Finite Automation）、Knuth-Morris-Pratt 算法（KMP Algorithm）、Boyer-Moore 算法、Simon 算法、Colussi 算法、Galil-Giancarlo 算法、Apostolico-Crochemore 算法、Horspool 算法和 Sunday 算法等。

基于本人的研究和理解，首先对其中三种基本的字符串匹配算法：朴素匹配算法（Naive Algorithm）、Rabin-Karp 算法、Knuth-Morris-Pratt 算法进行分析实现，再合理进行创新拓展实现 Boyer-Moore 算法。

朴素匹配算法

算法设计与分析说明

朴素匹配算法又称为暴力匹配算法（Brute Force Algorithm），是最原始也最容易想到的算法，通过一个循环将模式 P 和文本 T 进行字符逐一的匹配，可以形象的看成是用一个包含模式 P 的模板沿文本滑动，找到所有在范围 n-m+1 中存在满足条件$P[1..m]=T[s+1..s+m]$的有效位移 s。当某一个字符不相匹配时，将偏移量 s 加一，从头开始进行匹配。

算法设计思想实例说明

为形象具体的表达算法设计思想，通过如下的实例进行说明：

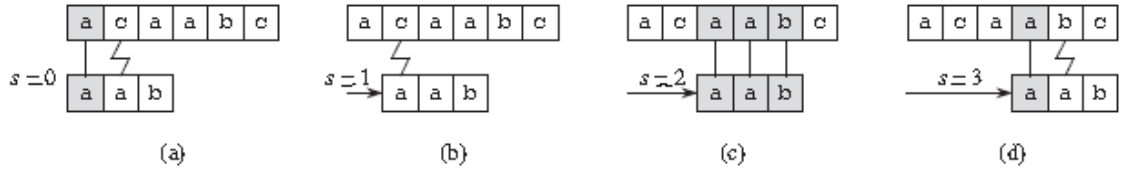


图 1

在图 1 中，对于模式 P = aab 和文本 T = acaabc，将模式 P 沿着 T 从左到右滑动，逐个比较字符是否相等以判断模式 P 在文本 T 中是否存在，若存在则返回偏移量 s 的值。

算法伪代码设计

根据如上的算法设计与分析以及实例说明，可以设计如下的伪代码，T 表示文本，P 表示模式，第 1-2 行首先求出文本及模式的长度，第 3-5 行通过一个循环，s 从 $[0..n-m]$ 逐一进行取值，表示偏移量，如果在某一个偏移值下满足 $P[1..m]=T[s+1..s+m]$，则表示匹配成功，并将偏移量 s 输出。

```
NAIVE-STRING-MATCHER(T, P)
1  n←length[T]
2  m←length[P]
3  for s←0 to n-m
4     do if P[1..m]=T[s+1..s+m]
5       then print "模式匹配的偏移值 s=" s
```

算法程序设计（C++）

```
void naive_matcher(string t,string p) //模式 p 和文本 t 字符串
{
    int n=t.length();
    int m=p.length();//分别求出模式和文本的字符串长度
    for(int s=0;s<=n-m;s++)//s 偏移量的循环
    {
       for(int i=0;i<m;i++) //模式串和文本进行字符的匹配
       {
            if(p[i]==t[s+i]&&i==m-1) printf("%d\n",s);
          //当字符全部匹配，即达到了模式串的长度
            else if(p[i]!= t[s+i]) break;
          //只要有一个字符不相等，就停止内循环，改变外循环的偏移量 s
        }`
    }
    return;//若不存在则返回-1
}
```

算法时间复杂度分析

通过分析易知，朴素匹配算法没有对模式 P 进行预处理，所以预处理的时

间为 0。通过外层的循环进行 n-m+1 次的偏移量 s 取值，对于每一个 s 取值，最坏要进行 m 次的字符相等判定，从而匹配的时间在最坏情况下为 $\theta((n-m+1)m)$，并且当 $m=\left\lfloor \frac{n}{2} \right\rfloor$，则为 $\theta(n^2)$。

Rabin-Karp 算法

算法设计与分析说明

通过分析朴素匹配算法，发现朴素算法会把前一次的匹配信息丢掉，然后从头再来，这样浪费资源，也增加了时间成本。由于完成两个字符串的匹配需要对其中包含的所有字符进行逐个比较，所需的时间较长，然而数值比较一次便可以判定是否相等。

假设文本 T 和模式 P 所涉及的字符集包含在 $\sum$ 中，并且假设每个字符都是以 d 为基数（进制）表示的数字，其中 $d=|\sum|$，例如由 ASCII 字符集 $\sum$ 内的字符构成的模式 P 和文本 T，因为 ASCII 字符集中有 128 个字符，则基数 d=128。

模式 $P[1..m]$，假设 p 表示其相应的十进制值，文本 $T[1..n]$，假设 t_s 表示长度为 m 的字符串 $T[s+1..s+m]$ 所对应的十进制值，其中 $s=0,1,...,n-m$。那么根据算法设计思想，当 $T[s+1..s+m]=P[1..m]$ 时，$t_s=p$。那么，计算出所有的 $t_0,t_1,t_2,...,t_{n-m}$，通过比较 p 和每一个 t_s 值，就能找到所有的有效偏移 s。

为计算所有的 t_s 和 p，根据霍纳法则【】，可以在 $\theta(m)$ 内计算出 p：

$$p=P[m]+d\left(P[m-1]+d\left(P[m-2]+...+d\left(P[2]+dP[1]\right)...\right)\right) \quad (1.1)$$

同理，根据 $T[1..m]$，也可在 $\theta(m)$ 时间内计算出 t_0，那么再根据递推式 1.2，可根据 t_0 在常数的时间内计算出 $t_1,t_2,...,t_{n-m}$。

$$t_{s+1}=d\left(t_s-d^{m-1}T[s+1]\right)+T[s+m+1]\left(s=0,1,...,n-m\right) \quad (1.2)$$

对递推式的 1.2 的说明：t_s 减去 $d^{m-1}T[s+1]$ 即减去了高位数字，再把结果乘以 d 就得数字向左移动一个数位，然后加上 $T[s+m+1]$，则加入一个适当的低位数字。

那么根据以上的算法分析，能在 $\theta(\mathrm{m})+\theta(n-m+1)=\theta(n)$ 时间内找到所有的有效偏移 s。

以上的算法设计，当 m 过大时，对应的数值 p 和 t_s 会过大，导致溢出。处理的思想是求余，设采用 d 进制的字母表 $\{0,1,...,d-1\}$，选取一个素数 q 值，并使得 dp 在一个计算机字长内，调整递推式（1.2），使其能够对模 q 有效，式子变为：

$$t_{s+1}=\left(d\left(t_s-T[s+1]h\right)+T[s+m+1]\right)\bmod q\left(s=0,1,...,n-m\right) \quad (1.3)$$

其中，$h\equiv d^{m-1}(\bmod q)$，h 的值可以预先通过 O(m)次计算得到 。

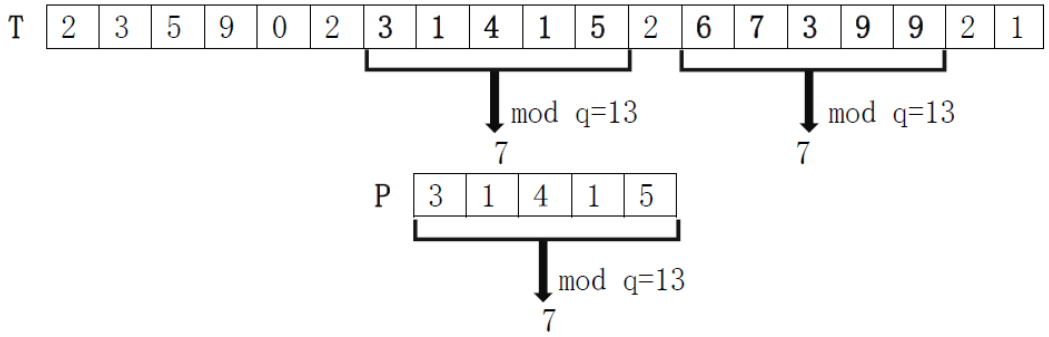
然而，引入求余的处理思想，当 $t_s\equiv p(\bmod q)$，并不能说明一定有 $t_s=p$。但是，当 $t_s\neq p(\bmod q)$，那么一定有 $t_s\neq p$。所以，当 $t_s\equiv p(\bmod q)$ 满足时，还需要进一步检测 $T[s+1..s+m]$ 和 $P[1..m]$ 每个字符逐一比较是否都相等，仅在每个字符都相等的情况下，才能最终判定 $T[s+1..s+m]=P[1..m]$。

根据上述的分析，可得表达式（1.3）：

$$\begin{cases} t_s\neq p & t_s\neq p(\bmod q) \\ t_s=p & t_s\equiv p(\bmod q)\ \&\&\ T[s+i]=P[i](i=1,2,...,m) \end{cases} \quad (1.3)$$

算法设计思想实例说明

为形象具体的表达算法设计思想，通过如下的实例进行说明：



假设 T=“23590231415267399921”，q=13，d=10，P=31415，m=5，发现当 s=6 时，满足 $T[s+1..s+5]=P[1..5]$，但是当 s=12 时，$T[s+1..s+5]=$"67399"，然而 $7\equiv 67399\equiv 31415(\bmod 13)$，但是字符串“67399”与字符串“31415”并不匹配。

算法伪代码设计

T 表示文本，P 表示模式，d 表示基数，即进制，和素数 q，第 3 行初始化模式的高位数值，第 4-8 行根据霍纳法则计算出模式 P 和 t_0 所对应的 d 进制的值，第 9-14 行循环偏移量 s 的值，在每一个 s 值下，当模式和文本对应的 d 进制相等时还需要进一步逐一检验文本 T 和模式 P 每一个字符是否都相等，只有相等时

才能最终说明$P[1..m]=T[s+1..s+m]$，并将偏移值输出，第 13-14 行，通过 t_0递推计算所有的$t_1,t_2,...,t_{n-m}$。

```
RABIN-KARP-MATCHER(T,P,d,q)
1  n=length[T]
2  m=length[P]
3  h=d^(m-1) mod q
4  p=0
5  t0=0
6  for i=1 to m
7     p=(dp+P[i])mod q
8     t0=(dt0+T[i])mod q
9  for s=0 to n-m
10    if p==ts
11       if P[1..m]==T[s+1..s+m]
12          print "模式匹配的偏移值 s=" s
13    if s<n-m
14       ts+1=(d(ts-T[s+1]h)+T[s+m+1])mod q
```

算法程序设计（C++）

```
//d 表示基数即进制，根据所有的字符集的大小而定，q 表示选取的素数
void rabin_karp_matcher(string t,string p,int d,int q)
{
    int n=t.length();
    int m=p.length();//分别求出模式和文本的字符串长度
    //预先处理好模式高位数 mod q 的值,h 的计算公式：pow(d,m-1) % q
    int h=1;
    for(int i=0;i<m-1;i++)
        h=(h*d)%q;
    //分别表示模式和文本的初始数值
    int P=0,T=0;
    //根据"霍纳法则"进行求解模式和文本 mod q 的值
    for(int i=0;i<m;i++)
    {
        P=(d*P+p[i])%q;
        T=(d*T+t[i])%q;
    }
    //偏移值进行循环，依次比较每一个偏移量 s 下是否满足字符串匹配
    for(int s=0;s<=n-m;s++)
    {
        //当模式和文本 hash 值相等，还需要进一步字符匹配验证
        if(abs(P-T)<0.0001)
        {
```

```
            for(int  j=0;j<m;j++) //模式串和文本进行字符的匹配
            {
                 if(p[j]==t[s+j]&&j==m-1) printf("%d\n",s);
                //当字符全部匹配，即达到了模式串的长度
                 else if(p[j]!= t[s+j]) break;
            //只要有一个字符不相等，就停止内循环，改变外循环的偏移量 s
             }
        }
        //根据 t0 进行递推所有的 t1,t2,t3,...,tn-m
        if(s<n-m)
        {
            T=(d*(T-t[s]*h)+t[s+m])%q;
            if(T<0)//当模为负值时，进行正数处理
                T=T+q;
        }
    }
    return;
}
```

算法时间复杂度分析

根据以上的算法设计和分析，依霍纳法则可以在$\theta(m)$的时间内计算出模式 p mod q 的值，以及在$\theta(n-m+1)$时间内计算出所有t_s mod q 的值，可是当$t_s = p$，并不一定能说明两个字符串就匹配，实质上就是 hash 值的冲突，可以通过增大素数 q 的值可以适当减少重复的 hash 值，从而可以减少时间消耗，所以在最坏的情况下，对于每一个 hash 值相等的$t_s = p$，都需要进行 m 次的字符匹配，则算法在最坏情况下的时间复杂度为$\theta((n-m+1)m)$，但一定的素数 q 下，总会使算法的时间复杂度小于$\theta((n-m+1)m)$。

KMP 算法

算法设计与分析说明

参考朴素匹配算法，对于每一次匹配，当进行到模式 P 和文本 T 某一个字符不相匹配时，偏移量 s 就加一，从头处理，可实际上，在一次匹配过程中，匹配成功的字符，其中包含的信息，能够推导出某一些偏移量 s 必定是无效的。对无效的 s 的判断可以减少很大的时间的开支，而这也恰恰是 KMP 算法的核心思想。

一次成功匹配 k 个字符的信息，可以通过模式 P 本身的前缀函数设为π来决定下一次的偏移量。假设模式$P[1..q]$与文本$T[s+1..s+q]$匹配，s'是最小的偏移

量，$s'>s$，那么对于某些$k<q$，满足$P[1..k]=T[s'+1..s'+k]$，其中$s'+k=s+q$，最小的偏移s'应当等于$s'=s+(q-k)$，其中$(q-k)$表示模式 P 的前缀长度范围的差值。在最好的情况下，$k=0$，因此$s'=s+q$，从而可以得到偏移$s+1,s+2,...,s+q-1$的值。在任何情况下，对于新的位移 s'无需把 P 的前 k 个字符与 T 中相应的字符进行比较，因为它们肯定匹配。

可以用模式 P 与其自身进行比较，以预先计算出这些必要的信息。引入模式的前缀函数 π，π包含有模式与其自身的位移进行匹配的信息。这些信息可用于避免在朴素匹配算法中，对无用位移进行测试。

$$\pi[q]=\max\{k:k<q\wedge P_k \lhd P_q\}$$

那么$\pi[q]$就代表了当前字符之前（包括当前字符）的字符串中，最长的共同前缀后缀的长度。

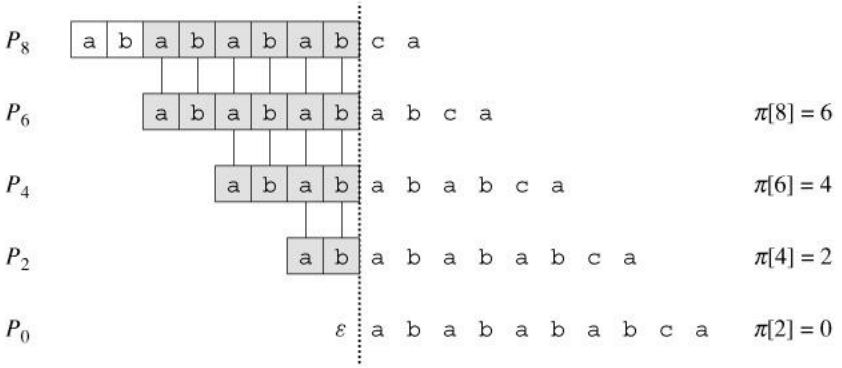
算法设计思想实例说明

为形象具体的表达算法设计思想，通过如下的实例进行说明：

假设模式 P = abababababca，如何计算 P 的前缀函数说明如下：

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)



(b)

1) π[1] = 0，a 仅一个字符，前缀和后缀为空集，共有元素最大长度为 0；
2) π[2] = 0，ab 的前缀 a，后缀 b，不匹配，共有元素最大长度为 0；
3) π[3] = 1，aba，前缀 a ab，后缀 ba a，共有元素最大长度为 1；
4) π[4] = 2，abab，前缀 a ab aba，后缀 bab ab b，共有元素最大长度为 2；
5) π[5] = 3，ababa，前缀 a ab aba abab，后缀 baba aba ba a，共有元素最大长度为 3；
6) π[6] = 4，ababab，前缀 a ab aba abab ababa，后缀 babab abab bab ab b，共有元素最大长度为 4；

7) π[7] = 5，abababa，前缀 a ab aba abab ababa ababab，后缀 bababa ababa baba aba ba a，共有元素最大长度为 5；
8) π[8] = 6，abababab，前缀 .. ababab ..，后缀 .. ababab ..，共有元素最大长度为 6；
9) π[9] = 0，abababab c，前缀和后缀不匹配，共有元素最大长度为 0；
10) π[10] = 1，abababab ca，前缀 .. a ..，后缀 .. a ..，共有元素最大长度为 1；

算法伪代码设计

输入文本 T 和模式 P，计算模式 P 本身的字符串信息，即最长前缀函数 π，第 5-12 行进行字符串匹配运算，第 6-7 行，当某一个字符匹配不相等时，在该位置，根据最长前缀函数 π 查找相应的值，从新的位置进行匹配，偏移值相当为 $s'=s'+(|q|-\pi(q))$。第 8-9 行，如果位置字符匹配，则 q+1 进行下一个字符的匹配。第 10-11 行，等到 q==m 时表明字符匹配完成，当下的偏移量即为 i-m，并对下一次匹配利用前缀函数求出初始位置。

对于求解一个字符串的最长前缀的函数，第 1-4 行，首先定义一个数组用于存放每一个字符包括自身的最长前缀数值。第 5-11 行，求解最长前后缀相等的函数数组。第 8-10 行，在每一个 q 下，验证上一次的最长前缀和后缀相等的位置的下一个字符是否和当下的字符相等即 P[k+1]!=P[q]，若相等，则证明最长前后缀相等的长度又增加了一，并将新的 k 值赋给当下字符表示的最长前缀值。第 6-7 行，当下一个字符和上一次循环的最长前缀数的下一个字符不相等时，回溯循环比较，进行再上一次的位置的最长前缀的位置是否和当下的字符相等，若都不相等，必定会到 k=0 的初始位置。

```
KMP-MATCHER(T,P)
1  n=length[T]
2  m=length[P]
3   π=COMPUTE-PREFIX-FUNCTION(p)
4  q=0
5  for i=1 to n
6     while q>0 and P[q+1]!=T[i]
7          q=π[q]
8     if P[q+1]==T[i]
9          q=q+1
10    if q==m
11         print "模式匹配的偏移值 s=" i-m
12         q=π[q]
COMPUTER-PREFIX-FUNCTION(P)
1  m=length[P]
2  let π[1..m] be a new array
3   π[1]=0
4  k=0
5  for q=2 to m
6      while k>0 and P[k+1]!=P[q]
7          k=π[k]
```

```
8       if P[k+1]==P[q]
9           k=k+1
10        π[q]=k
11  return π
```

算法程序设计（C++）

```
//用于存放模式P各字符的前缀函数的值
const int maxn=1000;
int pi[maxn];
//输入模式P计算其前缀函数
void compute_prefix_function(string p)
{
    int m=p.length(),k=0;
    //对于模式P的首个字符的前缀函数的值一定为0
    pi[0]=0;
    for(int q=1;q<m;q++)
    {
//当下一个字符，和上一个字符的前缀函数的值所在的下一个字符不相等时，回溯求解上一次的字符所对应的前缀函数的值
        while(k>0 && p[k]!=p[q])
            k=pi[k];
        //字符相等时，前缀函数加一
        if(p[k]==p[q])
            k++;
        pi[q]=k;
    }
    return;
}
//KMP算法求解字符串匹配
void kmp_matcher(string t,string p)
{
    int n=t.length();
    int m=p.length();
    //预先求解模式P的前缀函数
    compute_prefix_function(p);
    int q=0;
//遇到字符不相等时，下一个偏移的量的位置有当下字符的前缀函数而定
    for(int i=0;i<n;i++)
    {
        while(q>0 && p[q]!=t[i])
            q=pi[q];
        //字符相等，进行下一个字符的匹配
        if(p[q]=t[i])
            q++;
```

```
        if(q==m)
        {
//匹配完全以后 u，输出当下的偏移量，并将当下字符的下一个偏移量重新赋给 q，进行下一轮的循环
            printf("%d\n",i-m+1);
            q=pi[q];
        }
    }
    return;
}
```

算法时间复杂度分析

通过观察 KMP 匹配函数以及计算前缀函数的程序设计，可以发现程序的处理思想是大同小异的，在每一个字符进行匹配时，要么相等要么不相等，相等时就执行加一，进行下一个字符的匹配，如果匹配不相等时，文本的 i 不变（即 i 不回溯），模式串会跳过匹配过的 pi[q]个字符，减少无效的偏移量。所以，整个算法，如果文本串的长度为 n，模式串的长度为 m，那么匹配过程的时间复杂度为 $\theta(n)$，计算前缀函数的时间复杂度为 $\theta(m)$，KMP 匹配算法的整体时间复杂度为 $\theta(n)$。

创新拓展研究

Boyer-Moore 算法

算法设计与分析说明

在 1977 年，Robert S. Boyer (Stanford Research Institute) 和 J Strother Moore (Xerox Palo Alto Research Center) 共同发表了文章《A Fast String Searching Algorithm》，介绍了一种新的快速字符串匹配算法。这种算法在逻辑上相对于现有的算法有了显著的改进，它对要搜索的字符串进行倒序的字符比较，并且当字符比较不匹配时无需对整个模式串再进行搜索。【】

通过已经分析研究过的字符串匹配算法可知，在朴素匹配算法中，当发现模式 P 中的字符与文本 T 中的字符不匹配时，需要将文本 T 的比较位置向后滑动一位，模式 P 的比较位置归 0 并从头开始比较。而 KMP 算法则是根据计算最长前缀值的预处理的结果进行判断以使模式 P 的比较位置可以向后滑动多个位置。Boyer－Moore 算法在对模式 P 字符串的预处理过程为了达到相同效果，采用了两种不同的启发式方法。

Boyer－Moore 算法的两种启发式预处理方法	
坏字符(Bad Character Heuristic)	**定义：**当文本 T 中的某个字符跟模式 P 的某个字符不匹配时，称文本 T 中的这个失配字符为坏字符

	计算规则： ● 当字符失配时，后移位数=坏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，且如果坏字符不包含在模式串之中，则最右出现位置为-1 ● 当坏字符没出现在模式串中，等价于把模式串移动到坏字符的下一个字符 ● 当坏字符出现在模式串中，等价于把模式串第一个出现的坏字符和母串的坏字符对齐
好后缀(Good Suffix Heuristic)	**定义：**当文本 T 中的某个字符跟模式 P 的某个字符不匹配时，称文本 T 中的已经匹配的字符串为好后缀
	计算规则： ● 当字符失配时，后移位数=好后缀在模式串中当前的位置-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，则上一次出现的位置为-1 ● 当模式串中有子串匹配上好后缀，移动模式串，让该子串和好后缀对齐，如果超过一个子串匹配上好后缀，则选择最靠左边的子串对齐 ● 当模式串中没有子串匹配上好后缀，需要寻找模式串的一个最长前缀，并让该前缀等于好后缀的后缀，寻找到该前缀后，让该前缀和好后缀对齐 ● 当模式串中没有子串匹配上后后缀，并且在模式串中找不到最长前缀，让该前缀等于好后缀的后缀并直接移动模式到好后缀的下一个字符

Boyer－Moore 算法在预处理时，将为两种不同的启发法结果创建不同的数组，分别称为 Bad-Character-Shift（坏字符数组）和 Good-Suffix-Shift（好后缀数组）。当进行字符匹配时，如果发现模式 P 中的字符与文本 T 中的字符不匹配时，将比较两种不同启发法所建议的移动位移长度，并选择最大的一个值来对模式 P 的比较位置进行滑动。

$$s = s + \max(badchar[i], goodsuffix[i])$$

算法设计思想实例说明

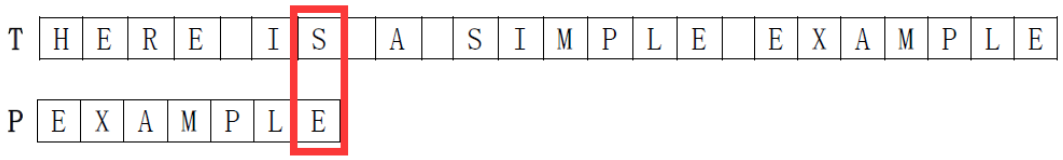
假设文本为 T，模式为 P，如下图所示：

T	H	E	R	E		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E

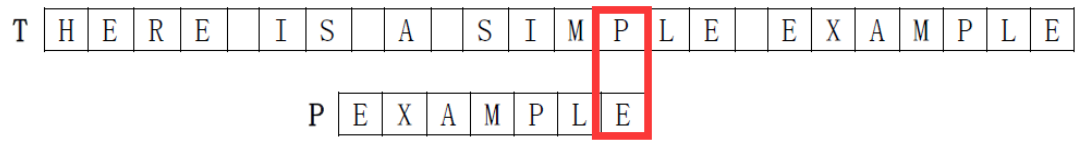
P	E	X	A	M	P	L	E

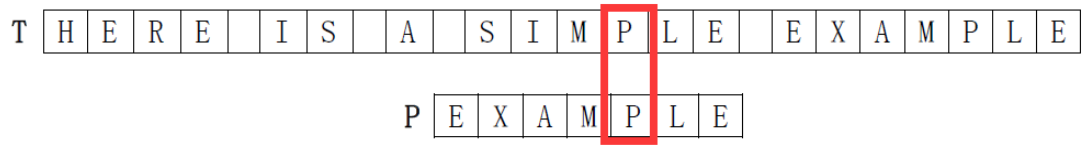
● T 与 P 头部对齐，从 P 尾部开始比较。由于 S 与 E 不匹配，S 就被称为 “坏字符”，即不匹配的字符，它对应着模式串 P 的第 6 位（从 0 开始）。因为 S

不包含在模式串 P 中（相当于最右出现位置是-1），通过计算可以把模式串后移 6-(-1)=7 位，从而直接移到 S 的后一位。

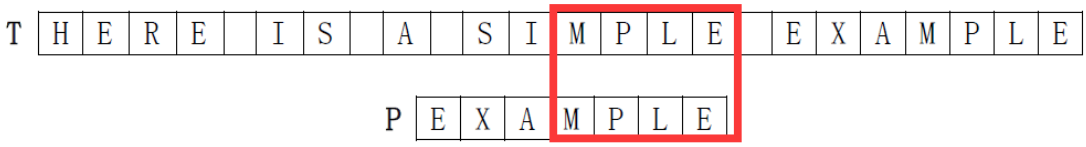


- 从尾部开始比较，发现 P 与 E 不匹配，所以 P 是“坏字符”。但 P 包含在模式串 P 之中，因为 P 这个“坏字符”对应着模式串的第 6 位，且在模式串中的最右出现位置为 4，所以，将模式串后移 6-4=2 位，两个 P 对齐。

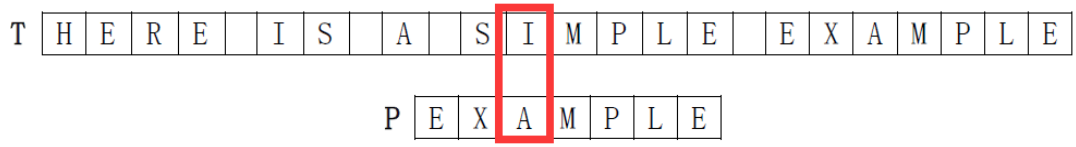


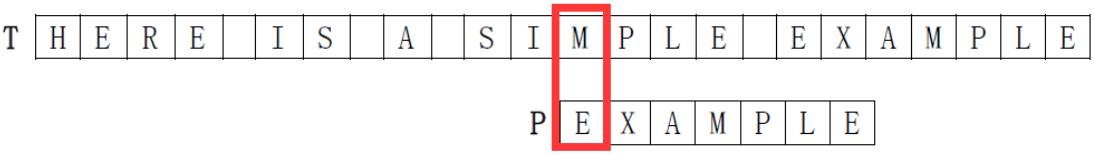


- 依次比较，得到 MPLE 匹配，称为“好后缀”，即所有尾部已经匹配的字符串，并且 MPLE、PLE、LE、E 都是好后缀。

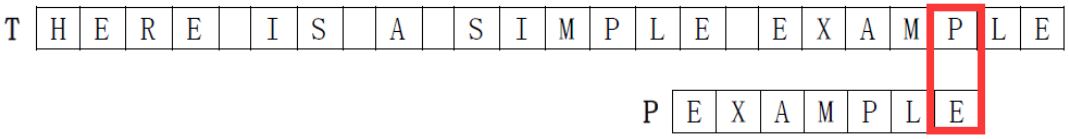


- 发现 I 与 A 不匹配：I 是坏字符。根据坏字符计算规则，此时模式串 P 应该后移 2-(-1)=3 位。

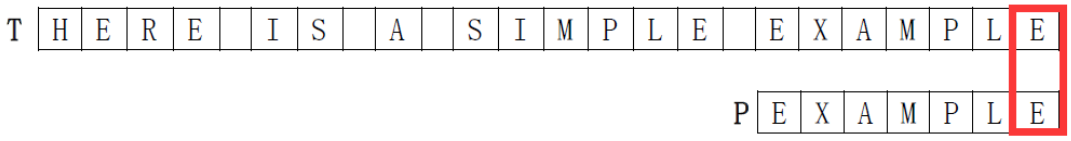




- 可是更优的移法是利用好后缀规则，当字符失配时，在所有的“好后缀”（MPLE、PLE、LE、E）之中，只有 E 在模式 P 的头部出现，根据好后缀计算规则，后移的位数为 6-0=6 位。通过比较可以看出，“坏字符规则”只能移 3 位，“好后缀规则”可以移 6 位。后移位数选择这两个规则之中的较大值。



- 继续从尾部开始比较，P 与 E 不匹配，因此 P 是“坏字符”，根据“坏字符规则”，后移 6 - 4 = 2 位。因为是最后一位失配，不能获得好后缀。



通过以上的步骤，文本 T 和模式 P 的匹配完成，可以看出 BM 算法根据坏字符和好后缀规则，使得失配时，文本串能够后移更多位，并且这两个规则的后移位数，只与模式 P 有关，与文本 T 无关。

算法程序设计（C++）

```
//定义所涉及的所有字符集的大小
#define ASCII_SIZE asize
const int maxn=1000,asize=256;
//用于存放坏字符和好后缀
int bad_char[maxn];
int good_suffix[maxn];
//构建坏字符数字
void buildbadchar(string p)
{
    int m=p.length();
//当坏字符没出现在模式串中，移动的距离就是模式串的长度
    for(int i=0;i<256;i++)
        bad_char[i]=m;
//计算模式 P 中每一个字符到串尾的距离
    for(int i=0;i<m-1;i++)
        bad_char[p[i]]=m-i-1;
    return;
}
//构建好后缀数组
void buildgoodsuffix(string p)
{
    int i,j,c,m=p.length();
    for(i=0;i<m-1;i++)
        good_suffix[i]=m;
//初始化模式 P 最末元素的好后缀值
    good_suffix[m-1]=1;
//循环找出模式 P 中各元素的好前缀值
    for(i=m-1,c=0;i>0;i--)
```

```
        for(j=0;j<i;j++)
        {
            if(p.substr(i,m-i)==p.substr(j,m-i))
            //if(memcmp(p+i,p+j,(m-i))==0)
            {
                if(j==0)
                    c=m-i;
                else
                    if(p[i-1]!=p[j-1])
                        good_suffix[i-1]=j-1;
            }
        }
    //根据模式P中个元素的前缀值，计算好后缀值
    for(i=0;i<m-1;i++)
    {
        if(good_suffix[i]!=m)
            good_suffix[i]=m-1-good_suffix[i];
        else
        {
            good_suffix[i]=m-1-i+good_suffix[i];
            if(c!=0 && m-1-i>=c)
                good_suffix[i]-=c;
        }
    }
}
//完整的BM匹配算法，根据好后缀和坏字符进行启发式移动位置，减少了无效的移动
void boyer_moore_matcher(string t,string p)
{
    int i,j;
    int n=t.length();
    int m=p.length();
    i=j=m-1;
    //构建好后缀和坏字符表
    buildbadchar(p);
    buildgoodsuffix(p);
    while(j<n)
    {
        //发现目标串与模式串从后向前第1个不匹配的位置
        while((i!=0)&&(p[i]==t[j]))
        {
            i--,j--;
        }
       //找到一个匹配的情况
```

```
        if(i==0 && p[i]==t[j])
        {
            printf("%d\n",j);
            j+=good_suffix[0];
        }
        else
        //坏字符表用字典构建

    j+=good_suffix[i]>bad_char[t[j]]?good_suffix[i]:bad_char[t[j]];
        i=m-1;
    }
}
```

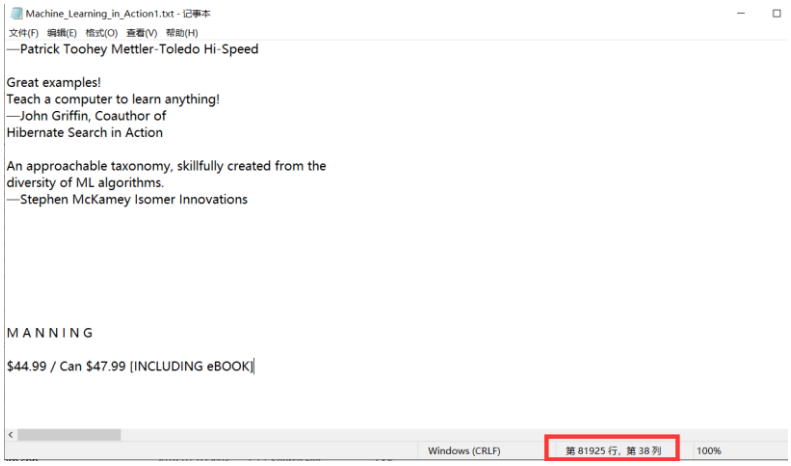
算法时间复杂度分析

BM 匹配算法根据好后缀和坏字符规则，使得字符失配时，文本串能够后移更多位，减少了不必要的无效偏移的操作，执行效率很高，并且通过相关论文的数学上的严格论证，该算法的时间复杂度为$\theta(n)$，其中 n 表示文本串 T 的长度。详细的推论过程，请见参考文献。

程序测试

测试用例的设计和选取

- 输入的测试用例文本 T 为一本纯英文的著作《Machine Learning in Action》，即机器学习实战，由于本书的主题离不开“machine”一词且存在大量的“machine”，所以，将“machine”作为模式 P 的输入测试用例，即作为搜索子串。根据算法运行求解出模式在文本中出现的所有的位置，以及出现的总次数，并检验所有的算法求解的结果是否相同，已验证算法实现的正确性。



Machine_Learning_in_Action.txt

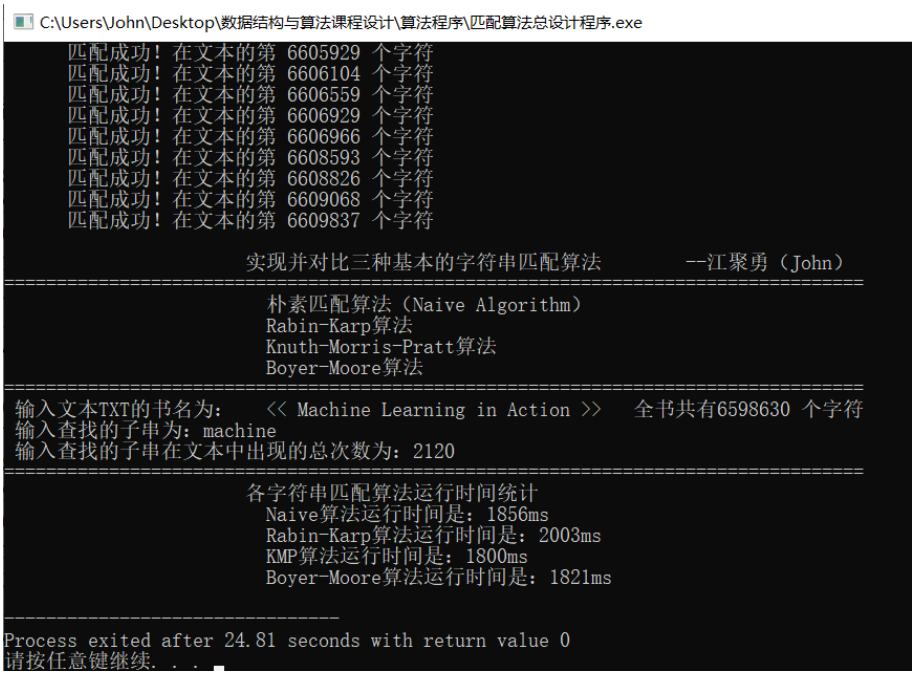
- 为了测试不同的字符串匹配算法，在不同数据规模下运行时间的变化，改变输入的文本 T 的字符量大小，以相同的模式 P=“machine”输入查找子串，记录不同数据规模下面的运行时间，并将结果利用 MATLAB 编程绘图，展示

不同数据规模下，算法运行时间大小的变化。

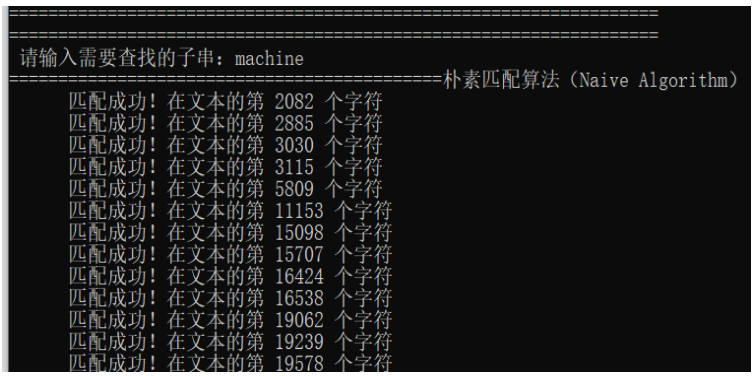
RunTime N	1000	10000	100000	1000000	10000000
Naive 算法					
Rabin-Karp 算法					
KMP 算法					
Boyer-Moor 算法					

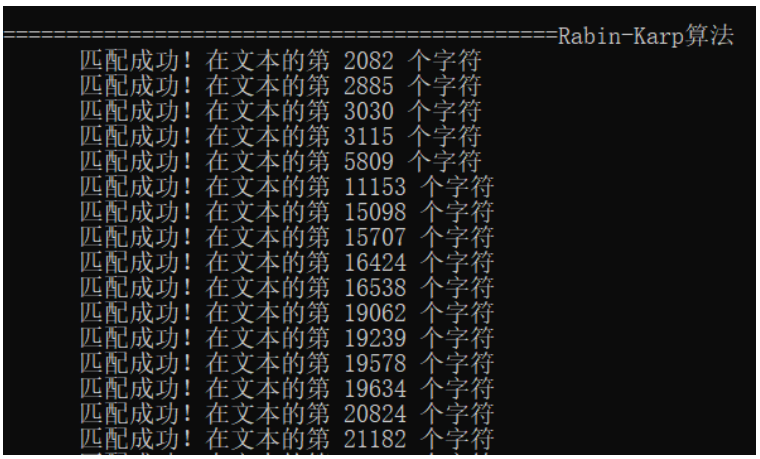
测试用例运行结果与分析

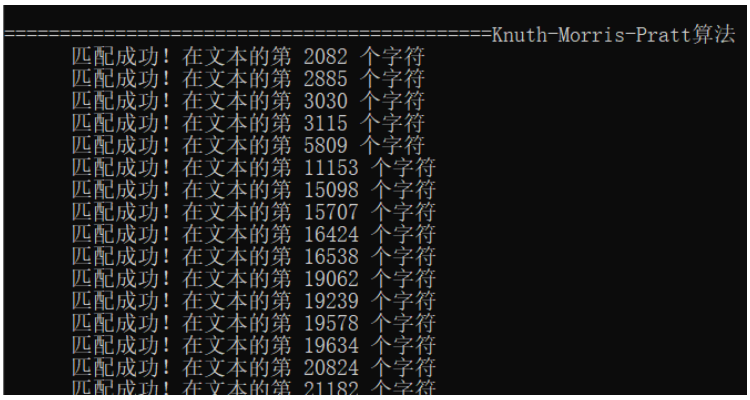
设计的所有字符串匹配算法以及所有算法的运行时间，以及文本 T 和模式 P 的信息的详细界面展示。

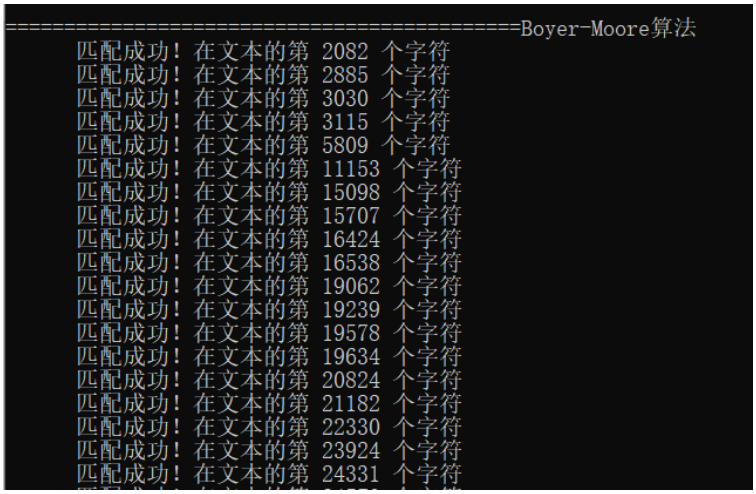


不同字符串匹配算法寻找相同子串所在文本中的位置结果：



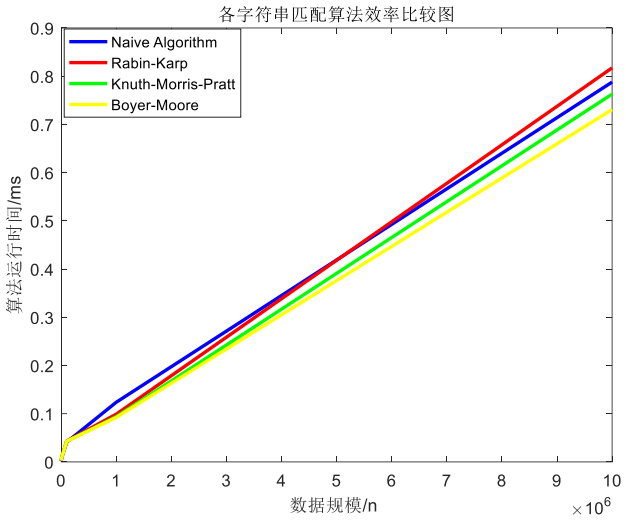






通过图中可以发现，所有算法的寻找子串的位置全部都相同，而且在文本 T 足够大的情况下，准确的定位了每一次模式 P 出现的位置。可以充分证明了，算法实现的正确性，以及查找子串的高效性。

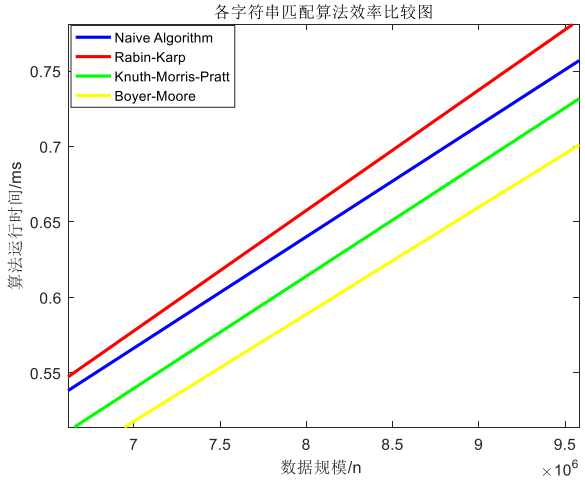
如下图是记录程序的运行结果，以后利用 MATLAB 绘制出的算法运行时间随文本字符个数的规模变化图，通过整体观察可以看出，所有的算法运行时间都是随着数据规模的增大而增大。并且通过放大数据规模很大以及数据规模很小的两部分，以便更加清楚的研究不同算法受数据规模的影响大小。



RunTime N	1000	10000	100000	1000000	10000000
Naive 算法	5	37	164	495	3151
Rabin-Karp 算法	12	36	173	396	3269
KMP 算法	14	36	171	375	3051
Boyer-Moor 算法	11	37	171	369	2923

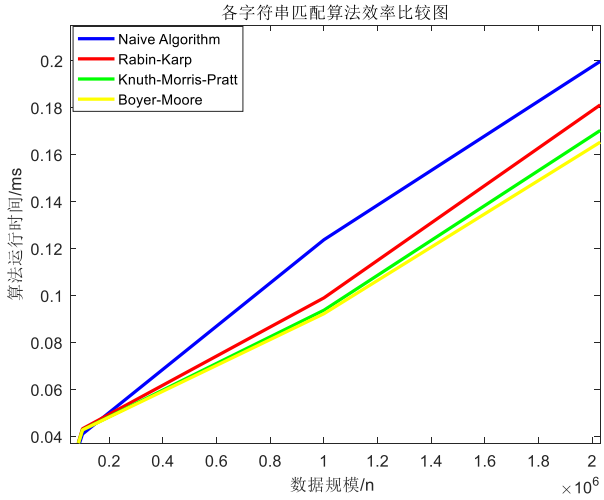
可观察到随着数据规模越来越大，算法运行所需时间：

Rabin-Karp 算法 > Naive 算法 > KMP 算法 > Boyer-Moore 算法



然而在数据规模较小时，后面两种算法依然呈现高效性，唯一变化的就是前两种算法的效率改变：

Naive 算法 > Rabin-Karp 算法 > KMP 算法 > Boyer-Moore 算法



总结

首先,实现了三种基本的字符串匹配算法,Naive 算法 、Rabin-Karp 算法 、KMP 算法，三种算法由一般的字符串匹配思想逐步的利用启发式的预处理方法，减少了不必要的无效移动操作，通过算法的分析和核心处理思想的理解，编写实现了相应的算法，并通过输入一个数据量非常大的文本进行测试，三种算法全部都通过了测试，并且输出了所有匹配的位置和次数。

进一步，通过创新研究 Boyer-Moore 算法，该算法利用好后缀和坏字符规则，进行启发式的预处理，大大减少了算法的执行时间，提高了匹配效率。在所有的算法中，同一文本 T 输入下，匹配相同的模式，Boyer-Moore 算法所需要的运行时间是最短的，并且随着数据量的越来越大，该算法表现的性能越来越好。

综合情况下可以得出，算法的效率分析就是：Naive 算法 < Rabin-Karp 算法 < KMP 算法 < Boyer-Moore 算法。对于不同算法的效率比较，我通过运用 MATLAB 编程绘制了算法的运行时间随数据规模的变化情况，清晰简明、直观，这也是我的创新之处。通过研究这些字符串匹配算法，让我对于一个问题思考的角度有了显著的不同,一个问题,可以通过多个巧妙地角度去达到相同的结果。Rabin-Karp 算法 、KMP 算法 和 Boyer-Moore 算法效率高，无疑就是进行了数据预处理，减少了无效的偏移值，从而提高算法的效率。在编辑文本程序过程中，经常需要在文本中找到某个模式的所有出现的位置。典型的情况就是，一段正在被编辑的文本构成一个文件，而所要搜索的模式是用户正在输入的特定的关键字，有效的解决这个问题的算法就是字符串匹配算法,该算法能够极大提高编辑文本程序时的响应效率。在其他很多应用中，字符串匹配算法用于在 DNA 序列中搜索特定的序列。在网络搜索引擎中也需要用这种方法来找到所需要查询的网页地址。

限于时间的原因，本来计划完成有限自动机算法（Finite Automation）、Horspool 算法和 Sunday 算法以进一步综合起来体现字符串匹配算法的灵活性以及巧妙性的。因为，这些算法都是字符串匹配算法中最为高效的一些算法，虽然 KMP 以及 BM 算法相对来说已经挺高效了，但是 Horspool 算法和 Sunday 算法还可以对 BM 算法进行进一步的优化和简化，使得算法的效率进一步的提升。此外，始终认为文本本身的字符排序和杂乱程度也会影响匹配的效率，所以尚未

解决，不同算法在不同的文本杂乱程度下，所能呈现的运行时间效率有什么不同。

通过这一次的课程设计，使我进一步的对字符串匹配算法有了更加深入的认识。对算法的设计和核心思想的处理以及实现能力，也有了提高。虽然，完成这样的一份课程设计，需要自学很多的额外的知识点，但是掌握了那些知识以后，确实对这个算法的认识有了进一步的提升。考虑到字符串匹配算法，其实在平时的文本排版时就会用上，可是一直思考过为什么能这样实现。也正是因为这样的一个课程设计，让我对于运用算法去解决实际中的问题，有了更加直观的感受，其实很多问题，其内核很可能就是基于自己所学过的知识的运用。

参考文献

[1]丁海军.数据结构(Java 语言版)[M].电子工业出版社，2015.

[2]严蔚敏，吴伟民. 数据结构（c 语言版）[M].北京：清华大学出版社，1997.

[3]Thomas H.Cormen，Charles E.Leiserson.算法导论(原书第 3 版)[M].机械工业出版社，2006.

[4]屈婉玲，刘田，张立昂，王捍贫.算法分析与设计(第二版)[M].北京：清华大学出版社，2011.

[5]Sun W, Manber U. A Fast Algorithm for Multi-pattern Searching.Tech. Rep. TR94-17, Department of Computer Science, University of Arizona, 1994-05.

[6]S. Wu, U. Manber, Fast text searching allowing errors, Communications of the ACM ,35(10) :83-91，1992.

[7]R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In N. J. Belkin and C. J. van Rijsbergen, editors, Proceedings of the 12th International Conference on Research and Development in Information Retrieval, pages 168-175. ACM Press, 1989.

[8]维基百科.Boyer-Moore 字符串搜索算法.

附录

MATLAB 绘图程序源代码（.m）

```
%% 测试数据规模
n=[1000 10000 100000 1000000 10000000]
%% 各算法所需要的运行时间
 y1=[5 37 164 495 3151]./4000% Naive Algorithm 算法运行的时间
 z1=[12 36 173 396 3269]./4000% Rabin-Karp 算法运行的时间
 h1=[14 36 171 375 3051]./4000% Knuth-Morris-Pratt 算法运行的时间
 t1=[11 37 171 369 2923]./4000%Boyer-Moore 算法运行时间
%% 绘制各算法运行时间随数据规模变化图
% Naive Algorithm 算法
 plot(n,y1,'b-','LineWidth',2);
 hold on;
% Rabin-Karp 算法
 plot(n,z1,'r-','LineWidth',2);
 hold on;
% Knuth-Morris-Pratt 算法
 plot(n,h1,'g-','LineWidth',2);
 hold on;
%Boyer-Moore 算法
 plot(n,t1,'y-','LineWidth',2);
 hold on;
 legend('Naive Algorithm','Rabin-Karp','Knuth-Morris-Pratt','Boyer-
Moore')
 xlabel('数据规模/n ')
 ylabel('算法运行时间/ms')
 title('各字符串匹配算法效率比较图')
```

字符串匹配算法完整源代码（C++）

```
#include<bits/stdc++.h>
using namespace std;
int naive_matcher(string t,string p) //模式 p 和文本 t 字符串
{
    int n=t.length();
    int m=p.length();//分别求出模式和文本的字符串长度
    int count=0;
    for(int s=0;s<=n-m;s++)//s 偏移量的循环
    {
        for(int i=0;i<m;i++) //模式串和文本进行字符的匹配
        {
```

```
            if(p[i]==t[s+i]&&i==m-1)
           {
              cout<<"          匹配成功！在文本的第 "<<s<<" 个字符
"<<endl;
              count++;
           }
           //当字符全部匹配，即达到了模式串的长度
            else if(p[i]!= t[s+i]) break;
           //只要有一个字符不相等，就停止内循环，改变外循环的偏移量 s

        }
    }
    return count;//若不存在则返回 0
}
//d 表示基数即进制，根据所有的字符集的大小而定，q 表示选取的素数
void rabin_karp_matcher(string t,string p,int d,int q)
{
    int n=t.length();
    int m=p.length();//分别求出模式和文本的字符串长度
    //预先处理好模式高位数 mod q 的值,h 的计算公式：pow(d, M-1) % q
    int h=1;
    for(int i=0;i<m-1;i++)
       h=(h*d)%q;
    //分别表示模式和文本的初始数值
    int P=0,T=0;
    //根据"霍纳法则"进行求解模式和文本 mod q 的值
    for(int i=0;i<m;i++)
    {
       P=(d*P+p[i])%q;
       T=(d*T+t[i])%q;
    }
    //偏移值进行循环，依次比较每一个偏移量 s 下是否满足字符串匹配
    for(int s=0;s<=n-m;s++)
    {
        //当模式和文本 hash 值相等，还需要进一步字符匹配验证
       if(abs(P-T)<0.0001)
       {
          for(int j=0;j<m;j++) //模式串和文本进行字符的匹配
          {
               if(p[j]==t[s+j]&&j==m-1)
                 cout<<"        匹配成功！在文本的第 "<<s<<" 个字符
"<<endl;
              //当字符全部匹配，即达到了模式串的长度
               else if(p[j]!= t[s+j]) break;
```

```
                //只要有一个字符不相等，就停止内循环，改变外循环的偏移量 s
            }
        }
        //根据 t0 进行递推所有的 t1,t2,t3,...,tn-m
        if(s<n-m)
        {
            T=(d*(T-t[s]*h)+t[s+m])%q;
            if(T<0)//当模为负值时，进行正数处理
                T=T+q;
        }

    }
    return;
}
//用于存放模式 P 各字符的前缀函数的值
const int maxn=10000;
int pi[maxn];
//输入模式 P 计算其前缀函数
void compute_prefix_function(string p)
{
    int m=p.length(),k=0;
    //对于模式 P 的首个字符的前缀函数的值一定为 0
    pi[0]=0;
    for(int q=1;q<m;q++)
    {
//当下一个字符，和上一个字符的前缀函数的值所在的下一个字符不相等时，回溯求解上一次的字符所对应的前缀函数的值
        while(k>0 && p[k]!=p[q])
            k=pi[k];
        //字符相等时，前缀函数加一
        if(p[k]==p[q])
            k++;
        pi[q]=k;
    }
    return;
}
//KMP 算法求解字符串匹配
void kmp_matcher(string t,string p)
{
    int n=t.length();
    int m=p.length();
    //预先求解模式 P 的前缀函数
    compute_prefix_function(p);
```

```
    int q=0;
//遇到字符不相等时，下一个偏移的量的位置有当下字符的前缀函数而定
    for(int i=0;i<n;i++)
    {
        while(q>0 && p[q]!=t[i])
            q=pi[q];
        //字符相等，进行下一个字符的匹配
        if(p[q]==t[i])
            q++;
        if(q==m)
        {
//匹配完全以后u，输出当下的偏移量，并将当下字符的下一个偏移量重新赋给q，进行下一轮的循环
            cout<<"          匹配成功！在文本的第 "<<i-m+1<<" 个字符"<<endl;
            q=pi[q];
        }
    }
    return;
}
#define ASCII_SIZE asize
const int asize=256;
int bad_char[maxn];
int good_suffix[maxn];
void buildbadchar(string p)
{
    int m=p.length();
//当坏字符没出现在模式串中，移动的距离就是模式串的长度
    for(int i=0;i<256;i++)
        bad_char[i]=m;
//计算模式P中每一个字符到串尾的距离
    for(int i=0;i<m-1;i++)
        bad_char[p[i]]=m-i-1;
    return;
}
void buildgoodsuffix(string p)
{
    int i,j,c,m=p.length();
    for(i=0;i<m-1;i++)
        good_suffix[i]=m;
//初始化模式P最末元素的好后缀值
    good_suffix[m-1]=1;
//循环找出模式P中各元素的好前缀值
    for(i=m-1,c=0;i>0;i--)
```

```
        for(j=0;j<i;j++)
        {
            if(p.substr(i,m-i)==p.substr(j,m-i))
            //if(memcmp(p+i,p+j,(m-i))==0)
            {
                if(j==0)
                    c=m-i;
                else
                    if(p[i-1]!=p[j-1])
                        good_suffix[i-1]=j-1;
            }
        }
    //根据模式P中个元素的前缀值，计算好后缀值
    for(i=0;i<m-1;i++)
    {
        if(good_suffix[i]!=m)
            good_suffix[i]=m-1-good_suffix[i];
        else
        {
            good_suffix[i]=m-1-i+good_suffix[i];
            if(c!=0 && m-1-i>=c)
                good_suffix[i]-=c;
        }
    }
}
void boyer_moore_matcher(string t,string p)
{
    int i,j;
    int n=t.length();
    int m=p.length();
    i=j=m-1;
    //构建好后缀和坏字符表
    buildbadchar(p);
    buildgoodsuffix(p);
    while(j<n)
    {
        //发现目标串与模式串从后向前第1个不匹配的位置
        while((i!=0)&&(p[i]==t[j]))
        {
            i--,j--;
        }
        //找到一个匹配的情况
        if(i==0 && p[i]==t[j])
        {
```

```
            cout<<"      匹配成功！在文本的第 "<<j<<" 个字符"<<endl;
            j+=good_suffix[0];
        }
        else
        //坏字符表用字典构建

    j+=good_suffix[i]>bad_char[t[j]]?good_suffix[i]:bad_char[t[j]];
        i=m-1;
    }
}
int main()
{
    int number=0;//用于统计模式在文本中出现的总次数
/*打开文件流，并将文件流按行读出到 buffer 中 ,然后从缓存中写入到字符串 file 中*/
    string buffer1,file1="";
    ifstream filestream1("Machine_Learning_in_Action1.txt");
    while(getline(filestream1,buffer1)) file1+=buffer1;
    int booklength=file1.length();
/*打开文件流，并将文件流按行读出到 buffer 中 ,然后从缓存中写入到字符串 file 中*/
    string buffer2,file2="";
    ifstream filestream2("Machine_Learning_in_Action2.txt");
    while(getline(filestream2,buffer2)) file2+=buffer2;
/*打开文件流，并将文件流按行读出到 buffer 中 ,然后从缓存中写入到字符串 file 中*/
    string buffer3,file3="";
    ifstream filestream3("Machine_Learning_in_Action3.txt");
    while(getline(filestream3,buffer3)) file3+=buffer3;
/*打开文件流，并将文件流按行读出到 buffer 中 ,然后从缓存中写入到字符串 file 中*/
    string buffer4,file4="";
    ifstream filestream4("Machine_Learning_in_Action4.txt");
    while(getline(filestream4,buffer4)) file4+=buffer4;
/*程序测试结果界面设计*/
cout<<"                              实现并对比三种基本的字符串匹配算法
"<<endl;
cout<<"============================================================
======"<<endl;
cout<<"                                 朴素匹配算法（Naive Algorithm）
"<<endl;
cout<<"                            Rabin-Karp 算法"<<endl;
cout<<"                            Knuth-Morris-Pratt 算法"<<endl;
cout<<"                            Boyer-Moore 算法"<<endl;
```

```
cout<<"============================================================
======"<<endl;
cout<<" 输入文本 TXT 的书名为：     << Machine Learning in Action >>
全书共有"<<booklength<<"个字符"<<endl;
cout<<file1<<endl;
cout<<"============================================================
======"<<endl;
cout<<"============================================================
======"<<endl;
cout<<" 请输入需要查找的子串：";
string keystr;
cin>>keystr;
cout<<"========================================= 朴 素 匹 配 算 法
（Naive Algorithm）"<<endl;
/*算法运行时间统计处理 */
    clock_t startrun1=clock();
    {
        number=naive_matcher(file1,keystr);
    }
    clock_t endrun1=clock();
    double                      runtime1=static_cast<double>(endrun1-
startrun1)/CLOCKS_PER_SEC*1000;
cout<<endl;

cout<<"=========================================Rabin-Karp 算 法
"<<endl;
/*算法运行时间统计处理 */
    clock_t startrun2=clock();
    {
        rabin_karp_matcher(file2,keystr,10,17);
    }
    clock_t endrun2=clock();
    double                      runtime2=static_cast<double>(endrun2-
startrun2)/CLOCKS_PER_SEC*1000;
cout<<endl;

cout<<"=========================================Knuth-Morris-
Pratt 算法"<<endl;
/*算法运行时间统计处理 */
    clock_t startrun3=clock();
    {
        kmp_matcher(file3,keystr);
    }
    clock_t endrun3=clock();
```

```
    double                      runtime3=static_cast<double>(endrun3-
startrun3)/CLOCKS_PER_SEC*1000;
cout<<endl;

cout<<"=========================================Boyer-Moore 算法
"<<endl;
/*算法运行时间统计处理 */
    clock_t startrun4=clock();
    {
        boyer_moore_matcher(file4,keystr);
    }
    clock_t endrun4=clock();
    double                      runtime4=static_cast<double>(endrun4-
startrun4)/CLOCKS_PER_SEC*1000;
cout<<endl;
cout<<"                          实现并对比三种基本的字符串匹配算法
--江聚勇（John）       "<<endl;
cout<<"=============================================================
======================"<<endl;
cout<<"                             朴素匹配算法（Naive Algorithm）
"<<endl;
cout<<"                          Rabin-Karp 算法"<<endl;
cout<<"                          Knuth-Morris-Pratt 算法"<<endl;
cout<<"                          Boyer-Moore 算法"<<endl;
cout<<"=============================================================
======================"<<endl;
cout<<" 输入文本 TXT 的书名为：     << Machine Learning in Action >>
全书共有"<<booklength<<" 个字符"<<endl;
cout<<" 输入查找的子串为："<<keystr<<endl;
cout<<" 输入查找的子串在文本中出现的总次数为："<<number<<endl;
cout<<"=============================================================
======================"<<endl;
/*算法运行时间汇总统计*/
cout<<"                                  各字符串匹配算法运行时间统计
"<<endl;
cout<<"                                       Naive 算法运行时间是：
"<<runtime1<<"ms"<<endl;//输出运行时间
cout<<"                                   Rabin-Karp 算法运行时间是：
"<<runtime2<<"ms"<<endl;//输出运行时间
cout<<"                                        KMP 算法运行时间是：
"<<runtime3<<"ms"<<endl;//输出运行时间
cout<<"                                  Boyer-Moore 算法运行时间是：
"<<runtime4<<"ms"<<endl;//输出运行时间
    return 0;
```

```
}
```