ISSN:0975-9646

Analysis of Pattern Matching Algorithms

Siva Kumar Kotamraju

CSE Department,
Vignans Nirula Institute of Technology & Science for Women
Pedapalakaluru, Guntur
Andhra Pradesh, India-522005

***Abstract*—The purpose of this paper is to analyze the efficiency of pattern matching algorithms. Brute-Force Algorithm, Knuth Morris Pratt Algorithm, Karp-Rabin Algorithm and Boyer Moore Algorithm are discussed here**

***Keywords*— Pattern Matching, String Matching**

I. INTRODUCTION

Finding all occurrences of a pattern in a text is a problem in text-editing applications. There a number of ways to find a string inside another string

II. METHOD 1: BRUTE FORCE METHOD

The first method is the Brute Force Method. The Brute Force Algorithm checks at all positions in the text between 0 and n ,for an occurrence of the pattern starts at that position. Then it shifts the pattern exactly one position to the right after each successful or unsuccessful attempt. As we go back and forth every time we have a mismatch in this method, we actually comparing every substring with length m and string with length n

Here is some working code:

```
#include <stdio.h>
void BF(char *x,char *y){

int i,j;
int m,n;
int count1=0;
i=0;
j=0;
n=strlen(y);
m=strlen(x);
while(j<n){
count1=0;
i=0;

while(x[i]==y[j]){
count1++;
i++;
j++;
if((i>=m) && (count1==m)){
count1=0;
printf("Match found at %d\n",j-i);
if(j>n-m)
break;
}
}
j=j-i+1;
}
}

int main(){
 char p[100]="ABC AAB";
char s[100]="GCABC AABAABC AAB GCDABCABB ABC AAB";
  BF(p,s);
return 0;
}
```

This is an Illustration of Brute Force Method

x[0]=A y[0]=G x[0]=A y[1]=C x[0]=A y[2]=A x[0]=A y[2]=A x[1]=B y[3]=B x[2]=C y[4]=C x[3]= y[5]= x[4]=A
y[6]=A x[5]=A y[7]=A x[6]=B y[8]=B
Match found at 2
x[0]=A y[3]=B x[0]=A y[4]=C x[0]=A y[5]= x[0]=A y[6]=A x[0]=A y[6]=A x[0]=A y[7]=A x[0]=A y[7]=A x[1]=B y[8]=B x[0]=A y[8]=B x[0]=A y[9]=A x[0]=A y[9]=A x[0]=A y[10]=A x[0]=A y[10]=A x[1]=B y[11]=B
x[2]=C y[12]=C x[3]= y[13]= x[4]=A y[14]=A x[5]=A y[15]=A x[6]=B y[16]=B
Match found at 10
x[0]=A y[11]=B x[0]=A y[12]=C x[0]=A y[13]= x[0]=A y[14]=A x[0]=A y[14]=A x[0]=A y[15]=A x[0]=A y[15]=A x[1]=B y[16]=B x[0]=A y[17]= x[0]=A y[18]=G x[0]=A y[19]=C x[0]=A y[20]=D x[0]=A y[21]=A x[0]=A
y[21]=A x[1]=B y[22]=B x[2]=C y[23]=C x[0]=A y[22]=B
x[0]=A y[23]=C x[0]=A y[24]=A x[0]=A y[24]=A x[1]=B
y[25]=B x[0]=A y[26]=B x[0]=A y[27]= x[0]=A y[28]=A x[1]=B y[29]=B x[2]=C y[30]=C x[3]= y[31]= x[4]=A y[32]=A x[5]=A y[33]=A x[6]=B y[34]=B
Match found at 28

III. METHOD2: KARP RABIN METHOD

Given text and pattern, the Karp Rabin Method has two stages. One is Pre-processing of the pattern and the second is searching phase for matching the pattern.

During preprocessing, hash value of the pattern, hx, is computed by adding ASCII value of the present character in the pattern to the twice of the previous sum. hy is computed by adding the ASCII value of the each character in the text of size m to the twice of previous sum.

During Searching Phase, the hash values hx and hy are compared, If they are equal and check substring of text of size m and pattern of size m are both equal. If they are also equal ,then a match is found and find the new hash value for substring of the given string. Repeat the searching phase with new hy and earlier computed hx in pre-processing phase and find all matching patterns

Here is some code that works

```
#include<stdio.h>
int hash(char *p){
int k,s=0,i=0,j=0;
k=strlen(p);
printf("Pattern length is %d\n",k);
s=p[i];
for(i=1;i<k;i++) {
s=s*2+p[i];
}
return s;
}

int hash2(char *t,int a,int b){
int i=a;
int s;
s=t[i];
for(i=a+1;i<b;i++){
s=s*2+t[i];
}
return s;
}

int compare(char *s1,char *s2,int m){
While(m--)
if(*s1==*s2)
return *s1-*s2;
else
s1++;
s2++;
return 0;
}

int search(char *t,char *p){
int j=0;
int n=strlen(t);
int m=strlen(p);
int hx,hy;
hx=hash(p);
while(j<=n-m){
hy=hash2(t,j,m+j);
if(hx==hy && compare(p,t+j,m)==0){
printf("\n Match found at %d\n",j);
}
++j;
}
}

int main(){
int res;
res=search("GCABC AABAABC AAB GCDABCABB ABC AAB","ABC AAB");
return 0;
}
```

ILLUSTRATION:

```
Pattern Length is 7
Pattern is ABC AAB
65 66 67 32 65 65 66 hx is 8056
71 67 65 66 67 32 65 hy is 8653
67 65 66 67 32 65 65 hy is 8283
65 66 67 32 65 65 66 hy is 8056
Match found at 2
66 67 32 65 65 66 65 hy is 7857
67 32 65 65 66 65 65 hy is 7331
32 65 65 66 65 65 66 hy is 6152
65 65 66 65 65 66 67 hy is 8275
65 66 65 65 66 67 32 hy is 8262
66 65 65 66 67 32 65 hy is 8269
65 65 66 67 32 65 65 hy is 8155
65 66 67 32 65 65 66 hy is 8056
Match found at 10
66 67 32 65 65 66 32 hy is 7824
67 32 65 65 66 32 71 hy is 7271
32 65 65 66 32 71 67 hy is 6033
65 65 66 32 71 67 68 hy is 8038
65 66 32 71 67 68 65 hy is 7821
66 32 71 67 68 65 66 hy is 7388
32 71 67 68 65 66 67 hy is 6395
71 67 68 65 66 67 65 hy is 8759
67 68 65 66 67 65 66 hy is 8496
68 65 66 67 65 66 66 hy is 8482
65 66 67 65 66 66 32 hy is 8292
66 67 65 66 66 32 65 hy is 8329
67 65 66 66 32 65 66 hy is 8276
65 66 66 32 65 66 67 hy is 8043
66 66 32 65 66 67 32 hy is 7798
66 32 65 66 67 32 65 hy is 7213
32 65 66 67 32 65 65 hy is 6043
65 66 67 32 65 65 66 hy is 8056
Match found at 28
```

IV. METHOD3:KNUTH-MORRIS-PRATT METHOD

The Knuth-Morris-Pratt algorithm is extension of the basic Brute Force Algorithm. Using KMP Algorithm, we stop looking back at the string, Therefore we use pre-computed data to skip forward as many characters as possible for the search to succeed but not just by 1 character.

Here is some working code:

```
#include <string.h>
#include<stdio.h>
void patternComputePrefixFunction(char *string, long len, long *prefix) {
long ki;
long qi;
prefix[0]=0;
ki=0;
for(qi=1;qi<len;qi++){
```

```
while((ki>0) && (string[ki]!=string[qi])) {
ki=prefix[ki-1];
}
If(string[ki]==string[qi]) {
ki++;
}
Prefix[qi]=ki;
}
}

int search(char  *x, unsigned char *y){
int i,j;
int count;
long m;
int n;
long *prefix;
m=strlen(x);
n=strlen(y);
/* Preprocessing*/
Prefix=(long *)malloc(100);
patternComputePrefixFunction(x,m,prefix);
/* Searching*/
count =0;
i=0;
j=0;
while(j<n){
count++;
while(i>0 && x[i]!=y[j]){
i=prefix[i-1];
}
if(x[i]==y[j]){
i++;
}
j++;
if(i>=m){
printf("\n Match found at %d\n",j-i);
i=prefix[i-1];
}
}
count=1;
return count;
}
int  main() {
int res;
res=search("ABC AAB","GCABC AABAABC AABGCDABCABB ABC AAB");
return 0;
}
```

This is how the comparison happens visually.

```
prefix[1]=0 prefix[2]=0 prefix[3]=0 prefix[4]=1
prefix[5]=1 prefix[6]=2
x[0]=A y[0]=G x[0]=A y[1]=C x[0]=A y[2]=A x[1]=B y[3]=B x[2]=C y[4]=C x[3]=  y[5]=  x[4]=A y[6]=A x[5]=A y[7]=A x[6]=B y[8]=B
Match Found at 2
i=0 x[0]=A y[9]=A
i=0 x[0]=A y[10]=A x[1]=B y[11]=B x[2]=C y[12]=C x[3]=  y[13]=  x[4]=A y[14]=A x[5]=A y[15]=A x[6]=B y[16]=B
Match found at 10
i=0 x[0]=A y[17]=  x[0]=A y[18]=G x[0]=A y[19]=C x[0]=A y[20]=D x[0]=A y[21]=A x[1]=B y[22]=B x[2]=C
y[23]=C
i=0 x[0]=A y[24]=A x[1]=B y[25]=B
i=0 x[0]=A y[26]=B x[0]=A y[27]=  x[0]=A y[28]=A x[1]=B y[29]=B x[2]=C y[30]=C x[3]= y[31]= x[4]=A y[32]=A x[5]=A y[33]=A x[6]=B y[34]=B
Match found at 28
```

V. METHOD 4:BOYER MOORE METHOD

The Boyer Moore Method is an efficient pattern search algorithm. It compares the characters to left from right in the text. The last occurrence of the character is calculated and when there is a mismatch, the position of index in string is shifted by the length minus index minus 1 or 1, whichever is maximum

Here is some working code

```
#include < stdio.h>
int Min(int a,int b){return a<b?a:b;}
int Max(int a,int b){return a>b?a:b;}
int last(char c,char *p)
{
char s[20];
char temp[20];
int i;
int j;
int k=1;
for(i=0;i<strlen(p);i++)
temp[p[i]]=strlen(p);
strcpy(s,p);
for(i=0;i<strlen(p);k=1,i++)
for(j=0;j<strlen(s);j++){
if(p[i]==s[j]){
k++;
if(k>=2 && p[i]==s[j]) {
temp[p[i]]=j;
}
}
}
for(i=0;i<strlen(p);i++)
if(c==p[i]){
k=Max(1,strlen(p)-temp[p[i]]-1);
break;
}
else
k=strlen(p);
return k;
}
int BM(char *s,char *q){
int i;
int j,k,r;
int n;
float count;
char *p;
char *t;
```

```
p=malloc(1000);
t=malloc(1000);
strcpy(p,s);
strcpy(t,q);
i=strlen(p)-1;
j=strlen(p)-1;
n=strlen(t);
do
{
if(count==0)
r=i;
if(p[j]==t[i]) {
count++;
if(j==0 && count==strlen(p)){
printf("\nMatch found at %d\n",i);
count=0;
i=i+strlen(p);
if(i>strlen(t))
break;
j=strlen(p)-1;
}
else {
i=i-1;
j=j-1;
}
}
else {
count=0;
i=r+last(t[r],p);
r=i;
if(i>strlen(t))
break;
j=strlen(p)-1;
}
}
while(i<n);
return 0;
}
int main(){
int res;
res=BM("ABC AAB","GCABC AABAABC AAB GCDABCABB ABC AAB");
return 0;
}
```

The following example illustrates this situation.

Example:

t[6]=A p[6]=B last is i=6 r=0 t[6]= A 7 t[7]=A p[6]=B last is i=7 r=7 t[7]= A 1 t[8]=B p[6]=B t[7]=A p[5]=A t[6]=A p[4]=A t[5]= p[3]= t[4]=C p[2]=C t[3]=B p[1]=B t[2]=A p[0]=A

Match found at 2

t[9]=A p[6]=B last is i=9 r=9 t[9]= A 1 t[10]=A p[6]=B last is i=10 r=10 t[10]= A 1 t[11]=B p[6]=B t[10]=A p[5]=A t[9]=A p[4]=A t[8]=B p[3]= last is i=8 r=11 t[8]= B 1 t[12]=C p[6]=B last is i=12 r=12 t[12]= C 4 t[16]=B p[6]=B t[15]=A p[5]=A t[14]=A p[4]=A t[13]= p[3]= t[12]=C p[2]=C t[11]=B p[1]=B t[10]=A p[0]=A

Match found at 10

t[17]= p[6]=B last is i=17 r=17 t[17]= 3 t[20]=D p[6]=B

last is i=20 r=20 t[20]= D 7 t[27]= p[6]=B last is i=27 r=27 t[27]= 3 t[30]=C p[6]=B last is i=30 r=30 t[30]= C 4

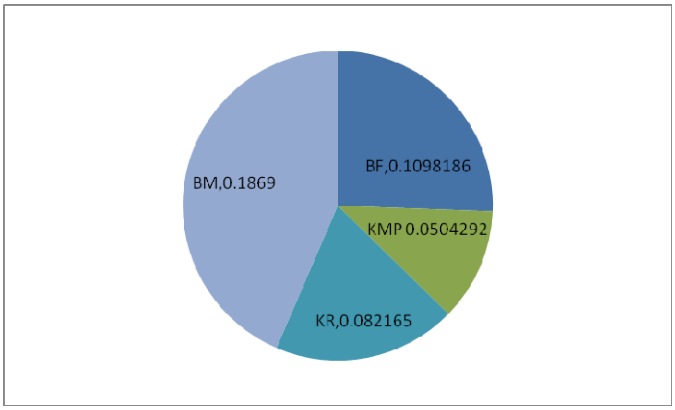
t[34]=B p[6]=B t[33]=A p[5]=A t[32]=A p[4]=A t[31]=

p[3]= t[30]=C p[2]=C t[29]=B p[1]=B t[28]=A p[0]=A

Match found at 28

VI. EXPERIMENT

An Experiment is done with the input as a textile and a pattern. The procedure is as follows: Get each line from the textfile as a string and search for the pattern in that string and repeat this process for all the remaining lines in the file till the End of File is reached.



VII. CONCLUSION

This is sufficient to know the basic Brute force algorithm and be aware of the other pattern matching methods. Here, we are concluding that KMP Algorithm is performing better compared to others.

REFERENCES

[1] INTRODUCTION TO ALGORITHMS, SECOND EDITION, T.H.COREMEN,C.E.LEISERSON, R.L.RIVEST AND C.STEIN, PHI

[2] BOYER, R. S., AND MOORE, J. S. (1977), A FAST STRING SEARCHING ALGORITHM, Comm. ACM 20,7.