Comparison of two-dimensional string matching algorithms

Chengguo Chang
Department of Information Engineering,
North China University of Water Resources and Electric Power
Zhengzhou, China
changchengguo@ncwu.edu.cn

Hui Wang
Department of Information Engineering,
North China University of Water Resources and Electric Power
Zhengzhou, China
wanghui@ncwu.edu.cn

***Abstract*—String matching is a special kind of pattern recognition problem, which finds all occurrences of a given pattern string in a given text string. The technology of two-dimensional string matching is applied broadly in many information processing domains. A good two-dimensional string matching algorithm can effectively enhance the searching speed. In this paper, the KMP algorithm, Rabin-Karp algorithm and their combinatorial are presented and compared, by a number of tests at diverse data scales, to validate the efficiency of these three algorithms.**

Keywords-string matching; KMP algorithm; Rabin-Karp algorithm

I. INTRODUCTION

The string matching is a basic problem in information technology, and is one of the most extensive problems in the computational complexity theory of algorithms. It is applied widely in the fields of text editing processing, image processing, document retrieval, natural language recognition, and biological sciences, etc. Furthermore, string matching is a most essential time-consuming question in those applications. Better string matching algorithms can significantly improve their application efficiencies. Therefore, research and design of a fast string matching algorithm has an important theoretical and practical significance.

II. TWO-DIMENSIONAL STRING MATCHING ALGORITHMS

A. Description of the problem

For a given $N * N$ digital character matrix $\boldsymbol{T}$, finding out all of the identical parts same as a given $M * M$ pattern matrix $\boldsymbol{P}$.

B. KMP Algorithm

The basic idea of KMP algorithm is checking upon $\boldsymbol{T}[i]$ and $\boldsymbol{P}[j]$ in the process of pattern matching, for the given text string $\boldsymbol{T}[n]$ and the pattern string $\boldsymbol{P}[m]$. If $\boldsymbol{T}[i] = \boldsymbol{P}[j]$, check upon $\boldsymbol{T}[i+1]$ and $\boldsymbol{P}[j+1]$. If $\boldsymbol{T}[i] \neq \boldsymbol{P}[j]$, there are two cases: if $j = 1$, move the text pointer i to the right next position, and then check upon $\boldsymbol{T}[i+1]$ and $\boldsymbol{P}[1]$; but if $1 < j \leq m$, slide the pattern j-***next***$[j]$ positions to the left, and then compare $\boldsymbol{T}[i]$ and $\boldsymbol{P}[\boldsymbol{next}[j]]$ (the array ***next*** is based on the partial matching of the pattern string itself). The repeated process will not terminate until $j = m$ or $i = n$.[5]

For this algorithm, firstly think about its single-string matching. For a single-string, the KMP algorithm could efficiently locate the pattern string in the target string, usually for $O(n)$ of time complexity[3]. But in the case of two-dimensional string, just like the situation we are discussing, the core idea is matching the top of the pattern matrix $\boldsymbol{P}$, then matching each column. When matching the top row, we could use the KMP algorithm. While in matching each column, this algorithm is not so much helpful. In general, instead, the column matchings are accomplished only by using the naive method.[1]

The KMP algorithm for row matching is listed as follows:

```
for (i=2;i<=m;i++) {
        while (j>0 && B[1][j+1]!=B[1][i])
            j=next[j];
        if (B[1][j+1]==B[1][i])
            ++j;
        next[i]=j;
 }
```

The naive method for column matching is:

```
void check(int x, int y) {
        for (int i=2;i<=M;i++)
                for (int j=1;j<=M;j++)
                        if (A[i+x-1][j+y-1]!=B[i][j])
                            return;
                        printf("%d %d\n",x,y);
}
 // Determination process
for (h=1;h<=N-M+1;h++) {
        j=0;
        for (i=1;i<=N;i++) {
                while (j>0 && B[1][j+1]!=A[h][i])
                    j=next[j];
                if (B[1][j+1]==A[h][i])
                    ++j;
                if (j==M)
                    check(h,i-M+1);
        }
}
```

978-0-7695-4647-6/12 $26.00 © 2012 IEEE
DOI 10.1109/ICCSEE.2012.29

C. *Rabin-Karp Algorithm*

The idea of Rabin-Karp algorithm is to take the string as a numeric value whose system base is equal to the length of the entire character set, then reach a conclusion from the comparisons between the corresponding values.[5] Given a text string ***T***[*n*] and a patter string ***P***[*m*], calculate an integer value corresponding to the pattern string P, and numeric values for all *m*-character substrings of the ***T***. As a consequent, we can see whether the pattern are matched or not with any of the substrings by comparing their values.

For example, for a given character set ∑ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, the length of ∑ is *d*=10. So every substring of the ∑ character set can be seen as a decimal number. Here, Both ***P*** and ***T*** are supposed to be strings based on the character set of length | ∑ | = *d*.

Given the pattern string ***P***[*m*], let *p* denote its corresponding value. Similarly, given the text string ***T***[*n*], the values of the *m*-length substrings of the ***T***[*n*] is denoted by t_s, Therefore, t_s corresponds to ***T***[*s*-1]~***T***[*s*+*m*-1] (here, 0 <= *s* <= *n*-*m*-1). [4]

The value ***P*** is evaluated by the following formula:

$$P = P[m] + d * (P[m-1] + d * (P[m-2] + ...)) \quad (1)$$

The value t_0 is also calculated in the same way and selected as the beginning. Then, we can evaluate t_{s+1} from t_s in the following recursive equation:

$$t_{s+1} = T[s+m] + d * (t_s + d_{m-1} * T[s]) \qquad (2)$$

From the above equation, we can get t_0 in *O*(*m*), and are expected to compute all t_s in *O*(*n* - *m* + 1) of time complexity. We could output all the valid shifts in *O*(*m*) + O(*n* - *m* + 1) = *O*(*n*) by comparing ***P*** with each of the t_s (as soon as getting a t_s, we immediately compare it with ***P***, and judge the matching result).

To compute the remaining values t_1, t_2, …, t_{n-m} in *O*(*n* - *m*) time, we have observed that t_s+1 can be computed from t_s in a constant time. As a result, we can compute ***P*** and t_0 in *O*(*m*) time, then compute t_1, t_2, …, t_{n-m} and complete matching in *O*(*n*-*m*+1) time. [2]

The pseudo code for the above algorithm is listed as follows:

```
typedef long long s64;
typedef pair<int,int> Pair;
const int MAXN=3000;
const s64 A=1041041,B=13133,C=192043;
const s64  D=10163,MO=1741013;
#define update(item,use,MUL)
item=((s64)item*MUL+use)%MO;
vector<Pair> ans1,ans2;
int N,M;
int a[MAXN][MAXN],b[MAXN][MAXN];
s64 hash[MAXN];
void Rabin_Karp(const s64 &MUL1,const s64
                    &MUL2,vector<Pair> &ans){
  for(int i=0;i<M;i++)
    MULM1=MULM1*MUL1%MO,
    MULM2=MULM2*MUL2%MO;
  for(int j=0;j<M;j++){
    for(int i=0;i<M;i++)
      update(tmp,b[i][j],MUL1);
    update(hashb,tmp,MUL2);
  }
  for(int i=0;i<M-1;i++)
    for(int j=0;j<N;j++)
      update(hash[j],a[i][j],MUL1);
      for(int i=M-1;i<N;i++){
        for(int j=0;j<N;j++)
          update(hash[j],a[i][j],MUL1);
        if (i!=M-1)
          for(int j=0;j<N;j++){
              hash[j]=hash[j]-a[i-M][j]*MULM1%MO;
              hash[j]=(hash[j]%MO+MO)%MO;
          }
        for(int j=0;j<M-1;j++)
          update(now,hash[j],MUL2);
        for(int j=M-1;j<N;j++){
          update(now,hash[j],MUL2);
          if (j!=M-1){
            now=now-hash[j-M]*MULM2%MO;
            now=(now%MO+MO)%MO;
          }
          if (now==hashb)
              ans.push_back(make_pair(i,j));
        }
      }
}
```

D. *KMP ＋Rabin-Karp Algorithm*

In practical application, KMP algorithm is appropriate for single-string matching problem. The time complexity of preprocessing phase (calculation for the array ***next***) of the KMP algorithm is *O*(*m*), while the matching time complexity is *O*(*n*). For single-string matching, the Rabin-Karp algorithm's preprocessing time complexity (calculation for *d*-base values) is *O*(*m*), while the matching time complexity is *O*((*n*-*m*+1)**m*). Through the comparison for their time complexities, we can see that the KMP algorithm is better than the Rabin-Karp algorithm for most of one-dimensional matching cases.

Just as mentioned above, the KMP algorithm is hard to use with two-dimensional string matching. However, the Rabin-Karp algorithm can easily overcome the two or even more dimensional problem. In order to achieve the best efficiency, we now consider the combination of these two algorithms for two-dimensioned strings matching problem, i.e. use the KMP algorithm for rows matching and the Rabin-Karp algorithm for columns.

The combinatorial pseudo code is as follows:

```
    typedef long long s64;
    typedef pair<int,int> Pair;
    const int MAXN=3000;
    const s64 A=1041041,B=13133,C=192043;
    const s64  D=10163,MO=1741013;
    #define update(item,use,MUL) item=(item*MUL+use)%MO;
    vector<Pair> ans1,ans2;
    int N,M;
    int a[MAXN][MAXN],b[MAXN][MAXN];
    s64 hash[MAXN],hashb[MAXN];
    int next[MAXN];
void kmp(s64 *str1,s64 *str2,vector<Pair> &ans,int ti){
        next[0]=-1;
        for(int i=1,j=-1;i<M;i++){
                while(j>=0 && str2[i]!=str2[j+1])
                        j=next[j];
                if (str2[i]==str2[j+1])
                        j++;
                next[i]=j;
        }
        for(int i=0,j=-1;i<N;i++){
                while(j>=0 && str1[i]!=str2[j+1])
                        j=next[j];
                if (str1[i]==str2[j+1])
                        j++;
                if (j==M-1){
                        ans.push_back(make_pair(ti,i));
                        j=next[j];
                }
        }
}
void Rabin_Karp(const s64 &MUL,vector<Pair> &ans){
        for(int i=0;i<M;i++)
           MULM=MULM*MUL%MO;
        for(int j=0;j<M;j++)
           hashb[j]=0;
        for(int i=0;i<M;i++)
          for(int j=0;j<M;j++)
             update(hashb[j],b[i][j],MUL);
        for(int i=0;i<M-1;i++)
          for(int j=0;j<N;j++)
             update(hash[j],a[i][j],MUL);
        for(int i=M-1;i<N;i++){
             for(int j=0;j<N;j++)
                 update(hash[j],a[i][j],MUL);
             if (i!=M-1)
                 for(int j=0;j<N;j++){
                    hash[j]=hash[j]-a[i-M][j]*MULM;
                    hash[j]=(hash[j]%MO+MO)%MO;
                 }
             kmp(hash,hashb,ans,i);
        }
}
```

III. Performance Comparison of the Three Algorithms

The sheet below illustrates the performance comparisons of the three pseudo codes, in the matching time (in seconds).

TABLE I. PERFORMANCE COMPARISON

Data scale	**KMP algorithm**	**Rabin-Karp algorithm**	**KMP+Rabin-Karp algorithm**
N=10，M=2	0.00	0.00	0.00
N=500，M=3	0.43	0.26	0.23
N=1000，M=500	2.10	1.04	0.98
N=1000，M=500	1.74	1.07	0.89
N=10000，M=10	70.22	32.23	27.12

The results were acquired by repeated running with the same data which was randomly generated by the program. According to the above results, we can see that the combination of the Rabin-Karp algorithm and the KMP algorithm significantly improve the matching speed, and therefore is suitable for two-dimensional matching problems.

IV. Conclusion

Through experimental verification of two-dimensional string matching processing, we can see that if the amount of data is small, the efficiency of KMP and Rabin-Karp algorithms are almost equal; but when the data set is large enough, the Rabin-Karp algorithm is obviously superior to the KMP algorithm, while the combination of them are the best. Generally, we can use the technique of the Rabin-Karp algorithm or the KMP algorithm + the Rabin-Karp algorithm for two-dimensional or multi-dimensional string matching problems. In practical applications, one of the three algorithms could be selected according to the actual data scale.

References

[1] Rujia Liu, Algorithm art and informatics competition. Beijing: Tsinghua University Press, pp 16-106, 2004

[2] Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest Clifford Ste , Introduction to algorithms, MIT Press, pp 794-799, 2005

[3] Weimin Yan, Data Structures , Beijing: Tsinghua University Press, pp 81-84, 2007

[4] “Data Structures”, USACO Training Gate

[5] JIng Liu, Computer algorithms introduction: design and analysis technology (1st Edition). Beijing: Science Press, pp 127-140, 2003.