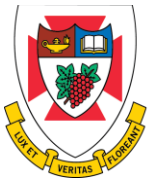


THE UNIVERSITY OF WINNIPEG

ADVANCE ALGORITHM DESIGN
GACS-7101

Project Report
Knuth Morris Pratt Algorithm

Submitted to:

Professor Dr.Yangjun Chen
Department Applied Computer Science
University of Winnipeg
Winnipeg, Manitoba, Canada

Submitted by:

Prateek Srivastava
Student ID # 3100767

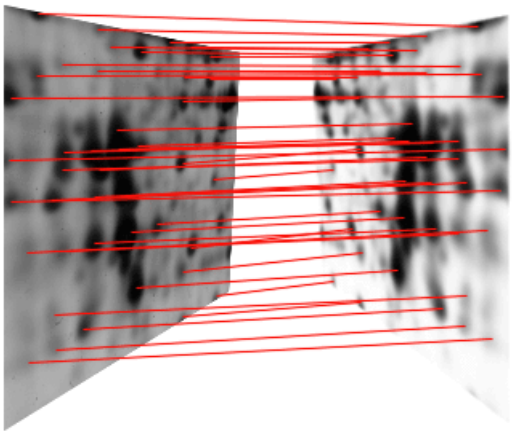
Submitted On:
04/16/2018

What is string matching?

The problem of finding the occurrences of a pattern string within another string.

eg: To identify certain gene

or cancer causing cell matching.



The algorithm was conceived in 1970 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris. The three published it jointly in 1977.[1] Independently, in 1969, Matiyasevich discovered a similar algorithm, coded by a two-dimensional Turing machine, while studying a string pattern-matching recognition problem.

Normal String Matching Complexity

- The performance is determined by how many comparisons are performed.
- Let m be the length of pattern and n be the length of text to be matched.
- The complexity is O(mn).
- Now the question is, can we do better?

Naive string matching:

```
for (i=0; T[i] != '\0'; i++)
    {
    for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;
    if (P[j] == '\0') found a match
    }
```

There are two nested loops; the inner one takes O(m) iterations and the outer one takes O(n) iterations so the total time is the product, O(mn). This is slow; we'd like to speed it up.

In practice this works pretty well -- not usually as bad as this O(mn) worst case analysis. This is because the inner loop usually finds a mismatch quickly and move on to the next position without going through all m steps. But this method still can take O(mn) for some inputs. In one bad example, all characters in T[] are "a"s, and P[] is all "a"'s except for one "b" at the end. Then it takes m comparisons each time to discover that you don't have a match, so mn overall.

How KMP is different?

- It uses a pre generated table called a 'Prefix Table'.
- A prefix table allow us to skip certain comparison.

Knuth, Morris and Pratt discovered first linear time string-matching algorithm by following a tight analysis of the naïve algorithm. Knuth-Morris-Pratt algorithm keeps the information that naïve approach wasted gathered during the scan of the text. By avoiding this waste of information, it achieves a running time of O($n + m$), which is optimal in the worst case sense. That is, in the worst case Knuth-Morris-Pratt algorithm we have to examine all the characters in the text and pattern at least once.

The difference is that KMP makes use of previous match information that the straightforward algorithm does not. In the example above, when KMP sees a trial match fail on the 1000th character (i = 999) because S[m+999] ≠ W[999], it will increment m by 1, but it will know that the first 998 characters at the new position already match. KMP matched 999 *A*characters before discovering a mismatch at the 1000th character (position 999). Advancing the trial match position m by one throws away the first *A*, so KMP knows there are 998

Algorithm

Prefix Table Psedocode

- M is the pattern (P) length
- i is the longest pattern that has been found in prefix
- j is current indicator.
- AR is the prefix table array.

KMP-Prefix (P)

begin

M ← [P]

AR[1] ← 0

i ← 0

for j=2 upto m step 1 do

while i>0 and P[i+1] != P[j] do

i ← AR[i]

If P[i+1]=P[j] then

i ← i+1

AR[j]= ←i

return AR

end

KMP Matcher Psedocode

- n is the text (T) length.
- m is the pattern (P) length.
- i points to the most matches in the pattern so far.
- j is the pointer.
- AR is the prefix table.
- KMP-Matcher(T,P)
- begin
- n ← [T]
- m ← [P]
- AR← KMP-Prefix(P)
- i ← 0
- for j=1 upto n step 1 do
- while i>0 and P[i+1] !=T[j] do
- i ← AR[i] (skip using prefix table)
- if P[i+1]=T[j] then
- i ← i+1 (next character matches)
- if i=m then
- Output (j-m) (pattern at shift to j-m position)
- i ← AR[i] (look for next match)
- end

Time Complexity of KMP Algorithm

- Since the two portions of the algorithm have, respectively, complexities of O(m) and O(n), the complexity of the overall algorithm is O(m + n).
- m is the length of pattern.
- n is the length of text.
- Best Case = O(n+m)

Because n is greater than m we can say O(n)

Space Complexity

- Space complexity= O(m)
- Where m is the length of prefix table.

C++ Code

```
#include <iostream>
#include <cstring>
using namespace std;
void preKMP(string pattern, int f[])
{
    int m = pattern.length(), k;
    f[0] = -1;
    for (int i = 1; i < m; i++)
    {
        k = f[i - 1];
        while (k >= 0)
        {
            if (pattern[k] == pattern[i - 1])
                break;
            else
                k = f[k];
        }
        f[i] = k + 1;
    }
}

//check whether target string contains pattern
bool KMP(string pattern, string target)
{
    int m = pattern.length();
    int n = target.length();
    int f[m];
```

```
28.     preKMP(pattern, f);
29.     int i = 0;
30.     int k = 0;
31.     while (i < n)
32.     {
33.         if (k == -1)
34.         {
35.             i++;
36.             k = 0;
37.         }
38.         else if (target[i] == pattern[k])
39.         {
40.             i++;
41.             k++;
42.             if (k == m)
43.                 return 1;
44.         }
45.         else
46.             k = f[k];
47.     }
48.     return 0;
49. }
50.
51. int main()
52. {
53.     string tar = "san and linux training";
54.     string pat = "lin";
55.     if (KMP(pat, tar))
56.         cout<<"""<<pat<<"' found in string '"<<tar<<"""<<endl;
57.     else
```

```
58.             cout<<"'"<<pat<<"' not found in string '"<<tar<<"'"<<endl;
59.         pat = "sanfoundry";
60.         if (KMP(pat, tar))
61.             cout<<"'"<<pat<<"' found in string '"<<tar<<"'"<<endl;
62.         else
63.             cout<<"'"<<pat<<"' not found in string '"<<tar<<"'"<<endl;
64.         return 0;
65.     }
```

Code Snapshot:

```cpp
#include <iostream>
#include <cstring>
    using namespace std;
    void preKMP(string pattern, int f[])
    {
        int m = pattern.length(), k;
        f[0] = -1;
        for (int i = 1; i < m; i++)
        {
            k = f[i - 1];
            while (k >= 0)
            {
                if (pattern[k] == pattern[i - 1])
                    break;
                else
                    k = f[k];
            }
            f[i] = k + 1;
        }
    }

    //check whether target string contains pattern
```

```cpp
    //check whether target string contains pattern
    bool KMP(string pattern, string target)
    {
        int m = pattern.length();
        int n = target.length();
        int f[m];
        preKMP(pattern, f);
        int i = 0;
        int k = 0;
        while (i < n)
        {
            if (k == -1)
            {
                i++;
                k = 0;
            }
            else if (target[i] == pattern[k])
            {
                i++;
                k++;
                if (k == m)
                    return 1;
```

```
            else
                k = f[k];
        }
        return 0;
    }

    int main()
    {
        string tar = "san and linux training";
        string pat = "lin";
        if (KMP(pat, tar))
            cout<<"'"<<pat<<"' found in string '"<<tar<<"'"<<endl;
        else
            cout<<"'"<<pat<<"' not found in string '"<<tar<<"'"<<endl;
        pat = "sanfoundry";
        if (KMP(pat, tar))
            cout<<"'"<<pat<<"' found in string '"<<tar<<"'"<<endl;
        else
            cout<<"'"<<pat<<"' not found in string '"<<tar<<"'"<<endl;
        return 0;
    }
```

Output

```
'lin' found in string 'san and linux training'
'sanfoundry' not found in string 'san and linux training'


...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment on KMP

- The Booth algorithm uses a modified version of the KMP preprocessing function to find the lexicographically minimal string rotation. The failure function is progressively calculated as the string is rotated.
- The algorithm uses a modified preprocessing function from the Knuth-Morris-Pratt string search algorithm. The failure function for the string is computed as normal, but the string is rotated during the computation so some indices must be computed more than once as they wrap around.

Time Complexity Comparisons

- Booth Algorithm.
- general formula is O(n) * (complexity_of_addition + complexity_of_shift), which most probably is, although very optimised, somewhere around O(n^2).
- KMP Algorithm.
- General formula is O(n).

References :

https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

https://pdfs.semanticscholar.org/8eaf/0670235200d9eee72e39e1b9a35fab96c179.pdf

http://www.cs.au.dk/~cstorm/courses/StrAlg_f13/slides/KMP-and-BM.pdf

https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/

https://brilliant.org/wiki/knuth-morris-pratt-algorithm/

https://dev.to/girish3/string-matching-kmp-algorithm-cie