Professor Anne Condon

Pattern and String Matching

CPSC 500
October 28th 2013
Scribe: Sedigheh Zolaktaf

1. Introduction

The object of string and pattern matching is to find all occurrences of a specific text pattern within a larger body of text. As with most algorithms, the main considerations for string searching are speed and efficiency. There are a number of string searching algorithms in existence today, but in this lecture we shall only review a naïve algorithm and the Knuth-Morris-Pratt algorithm.

We formalize the string-matching problem as follows. We assume that the text is an array T[1 .. n] of length n and that the pattern is an array P[1 .. m] of length $m \leq n$. We say that pattern P occurs with shift s in text T if $0 \leq s \leq n - m$ and P[1 .. m] = T[s+1 .. s+m]. If P occurs in T with shift s, then we call *s* a valid shift; otherwise, we call *s* an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.

Any algorithm for the string matching must examine every symbol in P and T at least once and requires $\Omega(m+n)$ time. We want to find an algorithm which matches this lower bound.

2. The naive string-matching algorithm

One of the most obvious approach towards the string matching problem would be to compare the first element of the pattern to be searched 'P', with the first element of the string 'T' in which to locate 'P'. If the first element of 'P' matches the first element of 'T', compare the second element of 'P' with second element of 'T'. If match found proceed likewise until entire 'P' is found. If a mismatch is found at any position, shift 'P' one position to the right and repeat comparison beginning from first element of 'T'. The naïve algorithm finds all valid shifts using a loop that checks the condition P[1 .. m] = T [s+1 .. s+m] for each of the n-m+1 possible values of s.

NAIVE-STRING-MATCHER(T,P)

```
1   n ← length[T]
2   m← length[P]
3   for s←0 to n-m
4       if P[1 .. m] = T[s+1 ..  s+m]
5           print "Pattern occurs with shift s"
```

Procedure NAIVE-STRING-MATCHER takes times O((n-m+1)m), and this is tight in the worst case. For each of the n-m+1 possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift. The worst case running time is thus Θ((n-m+1)m). The naïve string-matcher is inefficient because information gained about the text for one value of s is entirely ignored in considering other values of s.

3. The Knuth-Morris-Pratt algorithm

Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem. The algorithm keeps the information that naïve approach wasted during the scan of the text. By avoiding this waste of information, it achieves a running time of O(m+n). The implementation of Knuth-Morris-Pratt algorithm is efficient because it minimizes the total number of comparisons of the pattern against the input string. The key idea of this algorithm is to match P with a particular shift of T until mismatch occurs. Then it ***slides*** p to the right according to the sliding rule to avoid comparisons at shifts that are guaranteed not to result in a match. To illustrate the ideas of this algorithm, consider the following example:

T: xyxxyxyxyyxyxyxyyxyxyxxy
P: xyxyyxyxyxx

The KMP algorithm is similar to the naïve algorithm: it considers shifts in order from 1 to n-m, and determines if the pattern matches at that shift. The difference is that the KMP algorithm uses information gleaned from partial matches of the pattern and text to skip over shifts that are guaranteed not result in a match.

- P[1 .. 3] = T[1 .. 3] , P[4] != T[4]. Based on contents of P[1 .. 3], T[1 .. 3], how far forward can we slide the pattern? Slide forward 2 posotions. So that P[1] matches with T[3].
- Then compare P[2] and P[4]. P[2] != T[4], so slide forward 1.
- …..
- Later on P[1 .. 10] is matched with T[6 .. 15] and the dismatch P[11] != T[16] occurs. We ignore symbols of the pattern after position 10 and symbols of the text after position 15 and we can tell that the first possible shift that might result in a match is 12. Therefore, we will slide the pattern right, and next ask whether P [1 .. 11] = T[13 .. 23]. Thus, the next comparisons done are P[4] == T[16], P[5] == T[17], P[6] ==T[18] and so on, as long as matches are found.

3.1 Computing the prefix function, Π

The prefix function, Π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern 'P'. Let Π (q) be the length of the longest proper prefix of P that is also a proper suffix of P[1 .. q]. Π[1] is always equal to 0. Suppose we've calculated Π[1 .. i]. To compute Π[i+1] : let q= Π[i],

- If p[i+1]= p[q+1] then Π[i+1] = q+1
- Otherwise:
  set q = Π[q] and repeat test again unless q = 0 in which case Π[i+1] = 0.

Below is the pseudocode for computing the Π array.

COMPUTE-Π(P)

```
1    m← length[P]
2    Π[1] ← 0
3    for i ← 1to m -1            //calculate Π[i+1]
4        q ← Π[i]
5        while p[i+1]  != p[q+1] and q>0
6            q ← Π[q]
7        if p[i+1]  == p[q+1]
8            Π[i+1] ←q+1
9        else
10           Π[i+1] ←0
```

In the above pseudocode for computing the prefix function, the for loop runs 'm' times and the running time of computing the prefix functions is Θ(m).

Example: Below you can see the Π function computed for the given string P.

P:	x	y	x	y	y	x	y	x	y	x
q:	1	2	3	4	5	6	7	8	9	10
Π(q):	0	0	1	2	0	1	2	3	4	3

3.2 The Sliding rule

A prefix of a string S is any leading contiguous part of S. A suffix of the string S is any trailing contiguous part of S. For example, "x" and "xxx" are prefixes, and "xy" and "y" are suffixes of the string "xxxy". Also the empty string is considered prefix and suffix of all strings. A proper prefix and a proper suffix of a string are not equal to the string itself. In this algorithm we only consider proper prefix's and suffix's.

Suppose that P[1 .. q] is matched with T[i-q+1 .. i] and a mismatch then occurs: P[q+1] != T[i+1]. Then, the algorithm slides the pattern right so that the longest possible proper prefix of P[1 .. q] that is also a suffix of P[1 .. q] is now aligned with the text, with the last symbol of this prefix aligned at T[i]. If Π[q] is

the number such that P[1 .. Π[q]] is the longest proper prefix that is also a suffix of P[1 .. q], then the pattern slides so that P[1 .. Π[q]] is aligned with T[i- Π[q] + 1 .. i].

3.3 The KMP Matcher Algorithm

The KMP algorithm pre computes the values Π(q) and stores them in an array Π(1 .. m). The algorithm makes progress in two ways:

- P[q+1] = T[i+1] The length of the match is extended.
- P[q+1] != T[i+1] The pattern slides to the right.

The algorithm repeats these steps until the end of the text is reached. Both of these events can happen at most 2n times, thus the loop is executed at most 2n times and the cost of each iteration is a constant. Therefore, the running time of KMP is Θ(n). However, the overall complexity of the Knuth-Morris-Pratt algorithm is Θ*(m+n)* because the preprocessing of the array Π also has running time Θ(m). Below is the pseudocode for the KMP algorithm.

KMP(P, T)

```
1   n← length[T]
2   m ←length[P]
3   Π ← Compute- Π(p)
4   i ← 0
5   q ← 0
6   while i < n          // P[1 .. q]  = T[i-q+1 .. i]
7         if P[q+1] == T[i+1]
8               q ← q + 1
9               i ← i + 1
10              if q == m
11                  print "Pattern occurs with shift" i – m
12                  q ← Π[ q]
13        else
14              if q > 0
15                  q ← Π[ q]
16              else
17                  i ← i +1
```

4. Conclusion

In this lecture we reviewed two algorithms for solving the string-matching problem. The naïve string matcher algorithm is inefficient and runs in $\Theta(nm)$ time. After a shift of the pattern, the naive algorithm has forgotten all information about previously matched symbols. So it is possible that it re-compares a text symbol with different pattern symbols again and again.

The algorithm of Knuth, Morris and Pratt makes use of the information gained by previous symbol comparisons. It never re-compares a text symbol that has matched a pattern symbol. As a result, the complexity of the searching phase of the Knuth-Morris-Pratt algorithm is in $\Theta(n)$. However, a preprocessing of the pattern is necessary in order to analyze its structure. The preprocessing phase has a complexity of $\Theta(m)$. The overall complexity of the Knuth-Morris-Pratt algorithm is in $\Theta(m+n)$.

5. References

[1] Cormen, Thomas H. *Introduction to Algorithms*. Cambridge, Masachusetts: MIT, 2009.

[2] www.cs.ubc.ca/~hoos/cpsc445/Handouts/kmp.pdf

[3] www.cs.sjsu.edu/~mak/CS185C/W9Presentation.ppt