String Matching Algorithms

1. Naïve String Matching

The naïve approach simply test all the possible placement of Pattern P[1 . . m] relative to text T[1 . . n]. Specifically, we try shift s = 0, 1, . . . , n - m, successively and for each shift, s. Compare T[s +1 . . s + m] to
P[1 . . m].

```
NAÏVE_STRING_MATCHER (T, P)
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n - m do
4. if P[1 . . m] = T[s +1 . . s + m]
5.     then return valid shift s
```

The naïve string-matching procedure can be interpreted graphically as a sliding a pattern P[1 . . m] over the text T[1 . . n] and noting for which shift all of the characters in the pattern match the corresponding characters in the text.

In other to analysis the time of naïve matching, we would like to implement above algorithm to understand the test involves in line 4.

Note that in this implementation, we use notation P[1 . . j] to denote the substring of P from index i to index j. That is, P[1 . . j] = P[i] P[i +1] . . . P[j].

```
NAÏVE_STRING_MATCHER (T, P)
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n-m do
4.   j ← 1
5.   while j ≤ m and T[s + j] = P[j] do
6.     j ← j +1
7.   If j > m then
8.     return valid shift s
9. return no valid shift exist // i.e., there is no substring of T matching P.
```

Analysis

Referring to implementation of naïve matcher, we see that the for-loop in line 3 is executed at most n - m +1 times, and the while-loop in line 5 is executed at most m times. Therefore, the running time of the algorithm is O((n - m +1)m), which is clearly O(nm). Hence, in the worst case, when the length of the pattern, m are roughly equal, this algorithm runs in the *quadratic* time.

One worst case is that text, T, has n number of A's and the pattern, P, has (m -1) number of A's followed by a single B.

2. Knuth-Morris-Pratt Algorithm

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 Failure Function

The KMP algorithm preprocess the pattern P by computing a failure function f that indicates the largest possible shift s using previously performed comparisons. Specifically, the failure function $f(j)$ is defined as the length of the longest prefix of P that is a suffix of P[i . . j].

```
Input: Pattern with m characters
Output: Failure function f for P[i . . j]
KNUTH-MORRIS-PRATT FAILURE (P)
1. i ← 1
2. j ← 0
3. f(0) ← 0
4. while i < m do
5.   if P[j] = P[i]
6.     f(i) ← j +1
7.     i ← i +1
8.     j ← j + 1
9.   else if j > 0
10.    j ← f(j - 1)
11.  else
12.    f(i) ← 0
13.    i ← i +1
```

Note that the failure function f for P, which maps j to the length of the longest prefix of P that is a suffix of P[1 . . j], encodes repeated substrings inside the pattern itself.

As an example, consider the pattern P = a b a c a b. The failure function, $f(j)$, using above algorithm is

j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
$f(j)$	0	0	1	0	1	2

By observing the above mapping we can see that the longest prefix of pattern, P, is ″a b″ which is also a suffix of pattern P.

Consider an attempt to match at position i, that is when the pattern P[0 . . m -1] is aligned with text P[i . . i + m -1].

T : a b a c a a b a c c

P : a b a c a b

Assume that the first mismatch occurs between characters T[i+ j] and P[j] for 0 < j < m. In the above example, the first mismatch is T[5] = a and P[5] = b.

Then, T[i . . i + j -1] = P[0 . . j -1] = u

That is, T[0 . . 4] = P[0 . . 4] = u, in the example [u = a b a c a] and

T[i + j] ≠ P[j] i.e., T[5] ≠ P[5], In the example [T[5] = a ≠ b = P[5]].

When shifting, it is reasonable to expect that a prefix v of the pattern matches some suffix of the portion u of the text. In our example, u = a b a c a and v = a b a c a, therefore, 'a' a prefix of v matches with 'a' a suffix of u. Let l(j) be the length of the longest string P[0 . . j -1] of pattern that matches with text followed by a character c different from P[j]. Then after a shift, the comparisons can resume between characters T[i + j] and P[l(j)], i.e., T(5) and P(1).

```
T :  a  b  a  c  a  a  b  a  c  c
P :              a  b  a  c  a  b
```

Note that no comparison between T[4] and P[1] needed here.

```
Input: Strings T[0 . . n] and P[0 . . m]
Output: Starting index of substring of T matching P
KNUTH-MORRIS-PRATT (T, P)
1.  f ← compute failure function of Pattern P
2. i ← 0
3. j ← 0
4. while i < length[T] do
5.     if j ← m-1 then
6.      return i - m + 1 // we have a match
7.      i ← i + 1
8.      j ← j + 1
9.     else if j > 0
10.      j ←  f(j - 1)
11.     else
12.      i ← i + 1
```

Analysis

The running time of Knuth-Morris-Pratt algorithm is proportional to the time needed to read the characters in text and pattern. In other words, the worst-case running time of the algorithm is O(m + n) and it requires O(m) extra space. It is important to note that these quantities are independent of the size of the underlying alphabet.

3. Boyer-Moore Algorithm

The Boyer-Moore algorithm is consider the most efficient string-matching algorithm in usual applications, for example, in text editors and commands substitutions. The reason is that it woks the fastest when the alphabet is moderately sized and the pattern is relatively long.

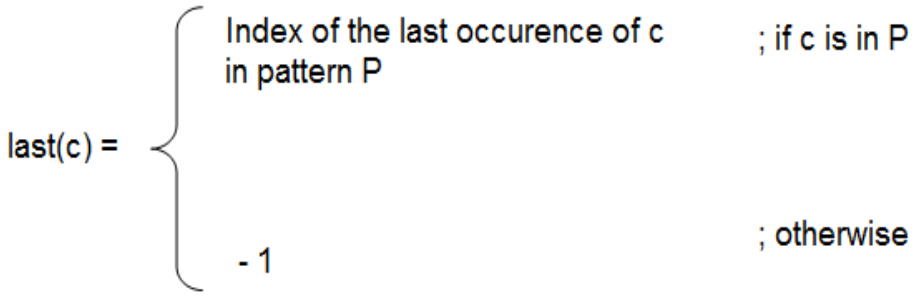
The algorithm scans the characters of the pattern from right to left beginning with the rightmost

character. During the testing of a possible placement of pattern P against text T, a mismatch of text character T[i] = c with the corresponding pattern character P[j] is handled as follows: If c is not contained anywhere in P, then shift the pattern P completely past T[i]. Otherwise, shift P until an occurrence of character c in P gets aligned with T[i].

This technique likely to avoid lots of needless comparisons by significantly shifting pattern relative to text.

Last Function

We define a function last(c) that takes a character c from the alphabet and specifies how far may shift the pattern P if a character equal to c is found in the text that does not match the pattern.



For example consider :

```
      0  1  2  3  4  5  6  7  8  9
T:    a  b  a  c  a  a  b  a  c  c

      0  1  2  3  4  5
P:    a  b  a  c  a  b
```

last(a) is the index of the last (rightmost) occurrence of 'a' in P, which is 4.

last(c) is the index of the last occurrence of c in P, which is 3

'd' does not exist in the pattern there we have last (d) = -1.

c	a	b	c	d
last(c)	4		3	-1

Now, for 'b' notice

```
T: a b a c a a b a c c
P: a b a c a b
```

Therefore, last(b) is the index of last occurrence of b in P, which is 5.

The complete last(c) function

c	a	b	c	d

last(c)	4	5	3	-1

Boyer-Moore algorithm

```
Input: Text with n characters and Pattern with m characters
Output: Index of the first substring of T matching P
BOYER_MOORE_MATCHER (T, P)
1. Compute function last
2. i ← m-1
3. j ← m-1
4. Repeat
5.    If P[j] = T[i] then
6.      if j=0 then
7.        return i // we have a match
10.     else
11.       i ← i -1
12.       j ← j -1
13.   else
14.     i ← i + m - Min(j, 1 + last[T[i]])
15.     j ← m -1
16. until i > n -1
17. Return "no match"
```

Analysis

The computation of the last function takes O(m+|∑|) time and actual search takes O(mn) time. Therefore the worst case running time of Boyer-Moore algorithm is O(nm + |∑|). Implies that the worst-case running time is quadratic, in case of n = m, the same as the naïve algorithm.

(i) Boyer-Moore algorithm is extremely fast on large alphabet (relative to the length of the pattern).

(ii) The payoff is not as for binary strings or for very short patterns.

(iii) For binary strings Knuth-Morris-Pratt algorithm is recommended.

(iv) For the very shortest patterns, the naïve algorithm may be better.

Source:

http://www.learnalgorithms.in/#