String matching

Boyer-Moore algorithm

Idea

The algorithm of Boyer and Moore [BM 77] compares the pattern with the text from right to left. If the text symbol that is compared with the rightmost pattern symbol does not occur in the pattern at all, then the pattern can be shifted by m positions behind this text symbol. The following example illustrates this situation.

Example:

```
0  1  2  3  4  5  6  7  8  9  ...
a  b  b  a  d  a  b  a  c  b  a
b  a  b  a  c
               b  a  b  a  c
```

The first comparison d-c at position 4 produces a mismatch. The text symbol d does not occur in the pattern. Therefore, the pattern cannot match at any of the positions 0, ..., 4, since all corresponding windows contain a d. The pattern can be shifted to position 5.

The best case for the Boyer-Moore algorithm is attained if at each attempt the first compared text symbol does not occur in the pattern. Then the algorithm requires only $O(n/m)$ comparisons.

Bad character heuristics

This method is called bad character heuristics. It can also be applied if the bad character, i.e. the text symbol that causes a mismatch, occurs somewhere else in the pattern. Then the pattern can be shifted so that it is aligned to this text symbol. The next example illustrates this situation.

Example:

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

Comparison b-c causes a mismatch. Text symbol b occurs in the pattern at positions 0 and 2. The pattern can be shifted so that the rightmost b in the pattern is aligned to text symbol b.

Good suffix heuristics

Sometimes the bad character heuristics fails. In the following situation the comparison a-b causes a mismatch. An alignment of the rightmost occurence of the pattern symbol a with the text symbol a would produce a negative shift. Instead, a shift by 1 would be possible. However, in this case it is better to derive the maximum possible shift distance from the structure of the pattern. This method is called good suffix heuristics.

Example:

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

The suffix ab has matched. The pattern can be shifted until the next occurence of ab in the pattern is aligned to the text symbols ab, i.e. to position 2.

In the following situation the suffix ab has matched. There is no other occurence of ab in the pattern.Therefore, the pattern can be shifted behind ab, i.e. to position 5.

Example:

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

In the following situation the suffix bab has matched. There is no other occurence of bab in the pattern. But in this case the pattern cannot be shifted to position 5 as before, but only to position 3, since a prefix of the pattern (ab) matches the end of bab. We refer to this situation as case 2 of the good suffix heuristics.

Example:

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

The pattern is shifted by the longest of the two distances that are given by the bad character and the good suffix heuristics.

Preprocessing for the bad character heuristics

For the bad character heuristics a function *occ* is required which yields, for each symbol of the alphabet, the position of its rightmost occurrence in the pattern, or -1 if the symbol does not occur in the pattern.

Definition: Let A be the underlying alphabet.

The <u>occurrence function</u> $occ : A^* \times A \longrightarrow \mathbb{Z}$ is defined as follows:

Let $p \in A^*$ with $p = p_0 \ldots p_{m-1}$ be the pattern and $a \in A$ an alphabet symbol. Then

$$occ(p, a) = \max\{ j \mid p_j = a \}$$

Here max($\varnothing$) is set to -1.

Example:

occ(text, x) = 2

occ(text, t) = 3

The rightmost occurence of symbol 'x' in the string 'text' is at position 2. Symbol 't' occurs at positions 0 and 3, the rightmost occurence is at position 3.

The occurrence function for a certain pattern p is stored in an array *occ* which is indexed by the alphabet symbols. For each symbol $a \in A$ the corresponding value $occ(p, a)$ is stored in $occ[a]$.

The following function *bmInitocc* computes the occurrence function for a given pattern p.

Bad character preprocessing

```
void bmInitocc()
{
    char a;
    int j;

    for (a=0; a<alphabetsize; a++)
        occ[a]=-1;

    for (j=0; j<m; j++)
```

```
        {
            a=p[j];
            occ[a]=j;
        }
    }
```

Preprocessing for the good-suffix heuristics

For the good-suffix heuristics an array s is used. Each entry $s[i]$ contains the shift distance of the pattern if a mismatch at position i – 1 occurs, i.e. if the suffix of the pattern starting at position i has matched. In order to determine the shift distance, two cases have to be considered.

Case 1: The matching suffix occurs somewhere else in the pattern (Figure 1).

Case 2: Only a part of the matching suffix occurs at the beginning of the pattern (Figure 2).

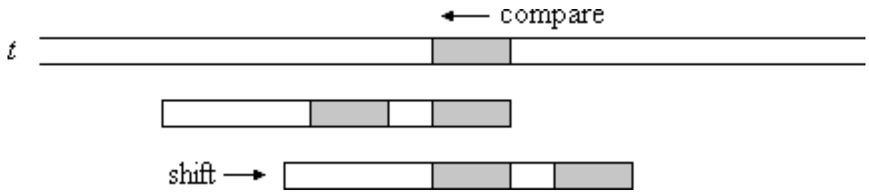

Figure 1: The matching suffix (gray) occurs somewhere else in the pattern

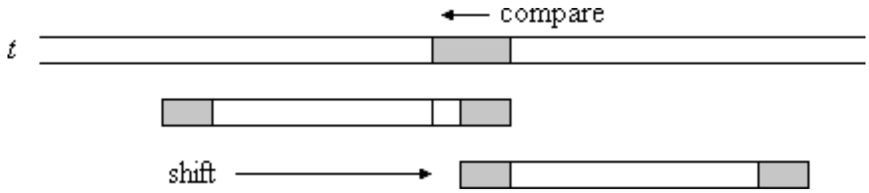

Figure 2: Only a part of the matching suffix occurs at the beginning of the pattern

Case 1:

The situation is similar to the Knuth-Morris-Pratt preprocessing. The matching suffix is a border of a suffix of the pattern. Thus, the borders of the suffixes of the pattern have to be determined. However, now the inverse mapping is needed between a given border and the shortest suffix of the pattern that has this border.

Moreover, it is necessary that the border cannot be extended to the left by the same symbol, since this would cause another mismatch after shifting the pattern.

In the following first part of the preprocessing algorithm an array f is computed. Each entry $f[i]$contains the starting position of the widest border of the suffix of the pattern beginning at position i. The suffix ε beginning at position m has no border, therefore $f[m]$ is set to m+1.

Similar to the Knuth-Morris-Pratt preprocessing algorithm, each border is computed by checking if a shorter border that is already known can be extended to the left by the same symbol.

However, the case when a border *cannot* be extended to the left is also interesting, since it leads to a promising shift of the pattern if a mismatch occurs. Therefore, the corresponding shift distance is saved in an array s – provided that this entry is not already occupied. The latter is the case when a shorter suffix has the same border.

Good suffix preprocessing case 1

```
void bmPreprocess1()
{
    int i=m, j=m+1;
    f[i]=j;
    while (i>0)
    {
        while (j<=m && p[i-1]!=p[j-1])
        {
            if (s[j]==0) s[j]=j-i;
            j=f[j];
        }
        i--; j--;
        f[i]=j;
    }
}
```

A visualization of the preprocessing algorithm is given in [3]. The following example shows the values in array *f* and in array *s*.

Example:

i:	0	1	2	3	4	5	6	7
p:	a	b	b	a	b	a	b	
f:	5	6	4	5	6	7	7	8
s:	0	0	0	0	2	0	4	1

The widest border of suffix babab beginning at position 2 is bab, beginning at position 4. Therefore, *f*[2] = 4. The widest border of suffix ab beginning at position 5 is ε, beginning at position 7. Therefore, *f*[5] = 7.

The values of array *s* are determined by the borders that cannot be extended to the left.

The suffix babab beginning at position 2 has border bab, beginning at position 4. This border cannot be extended to the left since *p*[1] ≠ *p*[3]. The difference 4 – 2 = 2 is the shift distance if bab has matched and then a mismatch occurs. Therefore, *s*[4] = 2.

The suffix babab beginning at position 2 has border b, too, beginning at position 6. This border cannot be extended either. The difference 6 – 2 = 4 is the shift distance if b has matched and then a mismatch occurs. Therefore, *s*[6] = 4.

The suffix b beginning at position 6 has border ε, beginning at position 7. This border cannot be extended to the left. The difference 7 – 6 = 1 is the shift distance if nothing has matched, i.e. if a mismatch occurs in the first comparison. Therefore, *s*[7] = 1.

Case 2:

In this situation, a part of the matching suffix of the pattern occurs at the beginning of the pattern. This means that this part is a border of the pattern. The pattern can be shifted as far as its widest matching border allows (Figure 2).

In the preprocessing for case 2, for each suffix the widest border of the pattern that is contained in that suffix is determined.

The starting position of the widest border of the pattern at all is stored in *f*[0]. In the example above this is 5 since the border *ab* starts at position 5.

In the following preprocessing algorithm, this value *f*[0] is stored initially in all free entries of array *s*. But when the suffix of the pattern becomes shorter than *f*[0], the algorithm continues with the next-wider border of the pattern, i.e. with *f*[*j*].

Good suffix preprocessing case 2

```
void bmPreprocess2()
{
    int i, j;
    j=f[0];
    for (i=0; i<=m; i++)
    {
        if (s[i]==0) s[i]=j;
        if (i==j) j=f[j];
    }
}
```

A visualization of the execution of the algorithm is given in [3]. The following example shows the final values of array *s*.

Example:

i:	0	1	2	3	4	5	6	7
p:	a	b	b	a	b	a	b	
f:	5	6	4	5	6	7	7	8
s:	5	5	5	5	2	5	4	1

The entire preprocessing algorithm of the Boyer-Moore algorithm consists of the bad character preprocessing and both parts of the good suffix preprocessing.

Boyer-Moore preprocessing

```
void bmPreprocess()
{
    int[] f=new int[m+1];
    bmInitocc();
    bmPreprocess1();
    bmPreprocess2();
}
```

Searching algorithm

The searching algorithm compares the symbols of the pattern from right to left with the text. After a complete match the pattern is shifted according to how much its widest border allows. After a mismatch the pattern is shifted by the maximum of the values given by the good-suffix and the bad-character heuristics.

Boyer-Moore searching algorithm

```
void bmSearch()
{
    int i=0, j;
    while (i<=n-m)
    {
        j=m-1;
        while (j>=0 && p[j]==t[i+j]) j--;
        if (j<0)
        {
            report(i);
            i+=s[0];
        }
        else
```

```
            i+=Math.max(s[j+1], j-occ[t[i+j]]);
    }
}
```

Analysis

If there are only a constant number of matches of the pattern in the text, the Boyer-Moore searching algorithm perfoms $O(n)$ comparisons in the worst case. The proof of this is rather difficult.

In general $\Theta(n \cdot m)$ comparisons are necessary, e.g. if the pattern is a^m and the text a^n. By a slight modification of the algorithm the number of comparisons can be bounded to $O(n)$ even in the general case.

If the alphabet is large compared to the length of the pattern, the algorithm performs $O(n/m)$ comparisons on the average. This is because often a shift by m is possible due to the bad character heuristics.

Conclusions

The Boyer-Moore algorithm uses two different heuristics for determining the maximum possible shift distance in case of a mismatch: the "bad character" and the "good suffix" heuristics. Both heuristics can lead to a shift distance of m. For the bad character heuristics this is the case, if the first comparison causes a mismatch and the corresponding text symbol does not occur in the pattern at all. For the good suffix heuristics this is the case, if only the first comparison was a match, but that symbol does not occur elsewhere in the pattern.

The preprocessing for the good suffix heuristics is rather difficult to understand and to implement. Therefore, sometimes versions of the Boyer-Moore algorithm are found in which the good suffix heuristics is left away. The argument is that the bad character heuristics would be sufficient and the good suffix heuristics would not save many comparisons. However, this is not true for small alphabets.

If for simplicity one wants to restrict oneself to the bad character heuristics, the Horspool algorithm [Hor 80] or the Sunday algorithm [Sun 90] are suited better.

References

[BM 77] R.S. Boyer, J.S. Moore: A Fast String Searching Algorithm. Communications of the ACM, 20, 10, 762-772 (1977)

[Hor 80] R.N. Horspool: Practical Fast Searching in Strings. Software - Practice and Experience 10, 501-506 (1980)

[Sun 90] D.M. Sunday: A Very Fast Substring Search Algorithm. Communications of the ACM, 33, 8, 132-142 (1990)

[Web 1] http://www-igm.univ-mlv.fr/~lecroq/string/

[Web 2] http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/stringmatchingclasses/BmStringMatcher.java
Boyer-Moore algorithm as a Java class source file

[Web 3] http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/bmPreprocess.xls
Boyer-Moore good suffix preprocessing visualization in Excel

Next: [Horspool algorithm] or ▲

H.W. Lang Hochschule Flensburg lang@hs-flensburg.de Impressum © Created: 22.02.2001 Updated: 29.05.2016