SOFTWARE-PRACTICE AND EXPERIENCE, VOL. 10, 501-506 (1980)

# Practical Fast Searching in Strings

R. NIGEL HORSPOOL

*School of Computer Science, McGill University, 805 Sherbrooke Street West, Montreal, Quebec H3A 2K6, Canada*

**SUMMARY**

**The problem of searching through text to find a specified substring is considered in a practical setting. It is discovered that a method developed by Boyer and Moore can outperform even special-purpose search instructions that may be built into the, computer hardware. For very short substrings however, these special purpose instructions are fastest-provided that they are used in an optimal way.**

KEY WORDS String searching Pattern matching Text editing Bibliographic search

## INTRODUCTION

The problem is that of searching a large block of text to find the first occurrence of a substring (which we will call the 'pattern'). This particular operation is provided in most text editing systems and it also has applications in bibliographic retrieval systems. Since the text to be searched can be overwhelmingly large – perhaps hundreds of thousands of characters – it is important to use efficient techniques.

Simple programs for searching text typically require a worst-case running time of $O(mn)$, where $m$ is the length of the pattern and $n$ is the length of the text. However, Knuth *et al.*[5] showed that this time can be reduced to $O(n)$ with a fairly complicated algorithm. Later, Boyer and Moore published a practical and simpler algorithm[1] that also has this linear worst case running time. In the average case, only a small fraction of the $n$ characters are actually inspected. (A recent paper by Galil[3] reports some improvements to the Boyer and Moore algorithm for its worst case behaviour.)

Many programmers may not believe that the Boyer and Moore algorithm (if they have heard of it) is a truly practical approach. It is the purpose of this paper to demonstrate that it is and to show the circumstances under which it should be employed. Many computers, particularly. the larger machines, possess instructions to search for individual characters within main memory. One might think that these instructions would permit codings of routines that could beat the Boyer and Moore algorithm. However, we will experimentally show that this is not always the case – even when the search instructions are used in the most efficient manner imaginable.

## EFFECTIVE USE OF A SEARCH INSTRUCTION

Some of the larger computers have a single instruction that can be used to search memory for the first occurrence of a designated character. The IBM 360-370 series (and the Interdata and

0038–0644/80/0610–0501$01.00 *Received* 18 *December* 1979

Amdahl computers with the same instruction set) has the Translate and test (TRT) instruction.[4] It can be programmed to search up to 256 bytes of memory for a particular character. The Burroughs B6500 has the Search While Not Equal (SNEU) instruction.[2] The UNIVAC 1100 series has the Search Equal (SE) instruction.[6]

If a search instruction is available, it would seem very reasonable to employ it in locating a substring within a longer string. We will present a simple and obvious method first. Our notation follows that of Boyer and Moore: STRING represents the text to be searched; STRINGLEN is its length. PAT is the substring we wish to find and PATLEN is its length. Additionally, we will use the notation S[I...J] to represent a substring consisting of the characters S[I], S[I+1] ... S[J]. We will call this algorithm SFC (Scan for First Character).

*Algorithm SFC*

```
if  patlen > stringlen then return 0;
ch ← pat[1];
i ← 0;
repeat
    scan string[i + 1 ... stringlen - patlen + 1] to find
       first occurrence of ch;
    if ch was not found then return 0;
    i ← position where ch was found;
until string[i... i + patlen-1] = pat;
return i;
```

If the algorithm returns 0 then PAT does not occur inside STRING; otherwise the result is the position of its first occurrence. The statement beginning 'Scan...' is to be implemented with the special search instruction. With the IBM 360-370 TRT instruction, we would have to code a short loop here to overcome the limited range of 256 bytes. We also note that the comparison of two strings in the loop termination condition can be performed with a single instruction on all the machines mentioned previously.

For practical applications, algorithm SFC does not use the search instruction in the best possible way. An elementary observation will make this clear. Suppose that STRING consists of (upper-case) English language text and that PAT is the word 'EXTRA'. The SFC algorithm scans for successive occurrences of the letter 'E' Unfortunately, 'E' is the most common letter in the English language and we would expect to hit an 'E' about every ten characters or so. Thus, there would usually be many unsuccessful comparisons between PAT and the text following each 'E' before obtaining the desired match.

On the other hand, 'X' is one of the least frequent letters in English. If we used the search instruction to locate successive occurrences of 'X' we would often be able to skip: through hundreds of characters at a time. By picking the character in PAT with the lowest frequency of occurrence in STRING, we can maximize the expected speed of our algorithm. This new method we call SLFC (Scan for Lowest Frequency Character).

*Algorithm SLFC*

```
if  patlen > stringlen then return 0;
find  j such that pat[j]  is the character in pat with the lowest frequency in English text;
ch ← pat[j];
i ← j - 1;
repeat
    scan string[i + 1 ... stringlen - patlen +1] to find
    the first occurrence of ch;
    if ch was not found then return 0;
    i ← position where ch was found;
 until string[i - j + 1 ... i + patlen - j] = pat;
 return i - j + 1 ;
```

The SLFC algorithm uses information that is not usually available to searching algorithms - namely character frequency information. We propose that this be provided in the form of a list of the possible characters sorted into order according to their expected frequency of occurrence. A perfect ordering would depend on the kind of text being searched. For example, with upper case text, 'E' is more frequent than 'T'. However, with mixed upper and lower-case text, the converse is true (because so many sentences begin 'The .. .') We contend that a perfect ordering is not really necessary. Even a random frequency ordering would give SFC and SLFC very similar performance. (SFC is the same as SLFC except that *j* is always chosen to be one.) Any improvement over the random ordering leads to superior performance by SLFC.

Table I: Expected number of characters that are skipped before finding the lowest frequency character in the pattern

| PATLEN | Expected distance | PATLEN | Expected distance |
|---|---|---|---|
| 1 | 94.0 | 7 | 401.3 |
| 2 | 161.8 | 8 | 440.5 |
| 3 | 218.8 | 9 | 478.1 |
| 4 | 270.0 | 10 | 514.2 |
| 5 | 316.4 | 11 | 549.2 |
| 6 | 360.1 | 12 | 583.0 |

The effect of the character frequency information is shown in Table I. For each value of PATLEN, the table shows the expected number of characters that we would expect to skip over when scanning for the lowest frequency character in PAT. To calculate these numbers, we have assumed that each character in STRING and in PAT is independently and randomly selected. Each character was given a selection probability that was determined by counting character frequencies in a large sample of text held in an on-line text-editing system. Clearly, the longer the pattern, the more characters we expect to skip. This is because longer patterns are more likely to contain a very low frequency character than a short pattern. Note that the table entry for PATLEN=1 effectively tells us how many characters are skipped in the SFC algorithms (regardless of the actual value of PATLEN).

## THE BOYER AND MOORE ALGORITHM

The basic Boyer and Moore searching algorithm can be written in the following form:

*Algorithm BM*

```
{ initialization of deltal and delta2 tables is omitted }
lastch. ← pat[patlen];
i ← patlen;
while i ≤ stringlen do
    begin
        ch ← string[i];
        if ch = lastch then
            begin
                j ← patlen - 1;
                repeat
                    if j = 0 then return i;
                    j ← j - 1;
                    i ← i - 1;
                until string[i] ≠ pat[j];
                i ← i + max(delta1[ch] , delta2[j]);
            end
        else
            i ← i + deltal[ch];
    end;
return 0;
```

There are two tables, DELTAl and DELTA2, whose entries are determined by analysis of the pattern. Most entries in DELTAl are equal in value to PATLEN. Consequently the algorithm usually advances through PATLEN characters at a time. The algorithm implemented by Boyer and Moore for their experiments was slightly different to the above. Their coding uses a third table. DELTA0, which is identical to DELTAl but for the DELTA[LASTCH] entry – it holds a very large integer so that the two tests `I<=`STRINGLEN and CH=LASTCH can be combined. This device is not entirely appropriate for computers with byte addressing, because DELTA0 cannot be implemented as a byte array (due to the single large valued entry) whereas DELTAl can be. Consequently, our experiments on the Amdahl computer (with IBM 370 architecture) used the version specified above.

In the normal usage of the algorithm, the DELTA2 table does not make much contribution to the overall speed. The only purpose of DELTA2 is to optimize the handling of repetitive patterns (such as 'XABCYYABC') and so to avoid a worst case running time of *O(mn)*. Since repetitive patterns are not too common, it is not worthwhile to expend the considerable effort needed to set up the DELTA2 table. To verify the unimportance of DELTA2, we constructed and timed the following simplified version of the Boyer and Moore algorithms. Our table, DELTA12 is the same as DELTAl except that the DELTA12[LASTCH] entry has its value taken from DELTA2[PATLEN].

*Algorithm SBM*

```
delta12[*] ← patlen; { initialize whole array }
for j ← 1 to patlen - 1 do
    delta12[pat[j]]. ←  patlen - j;
lastch ← pat[patlen];
i ← patlen;
while i ≤ stringlen do
    begin
        ch ← string[i];
        if  ch = lastch then
            if string[i - patlen + 1 … j] = pat then
                return i - patlen + 1 ;
            i ← i + delta12[ch];
    end;
return 0;
```

We note that an instruction for comparing character strings is useful for implementing the comparison between STRING and PAT.

## COMPARISON OF METHODS

The four different search algorithms were coded in *370/Assembler* as efficiently as possible. They were compared by timing them on the task of locating every occurrence of some pattern within a very large block of text (80,000 characters) and the timings were averaged over many repetitions of the task. The patterns were randomly selected substrings within the text.

Table II: Experimentally observed search rates
for the four algorithms under consideration

| | Search rate (millions chars/s) | | | |
|---|---|---|---|---|
| PATLEN | SFC | SLFC | BM | SBM |
| 2 | 3.3 | 4.0 | 2.1 | 2.4 |
| 3 | 4.3 | 4.8 | 3.3 | 3.2 |
| 4 | 3.8 | 5.2 | 4.3 | 4.6 |
| 5 | 3.6 | 5.3 | 5.0 | 5.2 |
| 6 | 3.8 | 5.4 | 6.9 | 7.1 |
| 7 | 5.4 | 5.4 | 7.0 | 7.4 |
| 8 | 3.6 | 5.3 | 7,5 | 8.1 |
| 9 | 4.6 | 5.2 | 8.5 | 8.7 |
| 10 | 5.1 | 5.7 | 8.5 | 7.9 |
| 11 | 3.5 | 5.3 | 9.4 | 9.8 |
| 12 | 3.8 | 5.6 | 9.4 | 9.4 |

The results are shown in Table II. This table gives the search rate in millions of characters per second as measured on the Amdahl V7 computer. There are several observations to be made about these numbers. First of all, BM and SBM give nearly identical timings – demonstrating the unimportance of the DELTA2 table in normal use. Secondly, SLFC is quite

superior to SFC (only for PATLEN equal to one would they be similar). The conclusion to be drawn from this is that the technique of searching for low frequency characters pays off handsomely. Thirdly, we see that BM (and SBM) has similar speed to SFC for PATLEN equal to five. For larger PATLEN values, SFC is inferior and for smaller PATLEN values it is superior.

There is one important factor that was not considered in our experiments. The timings do not include the work of initializing tables – on the assumption that we want to find the limiting speed of each algorithm (i.e. the speed when searching an infinite volume of text). In practice, the initialization code may be significant. We note that the DELTA2 table requires the most computation and this is another justification for using the SBM version of Boyer and Moore's algorithm.

## CONCLUSIONS

An important result is that we have demonstrated the Boyer and Moore algorithms to be an astonishingly fast method of searching text. For pattern lengths of six or greater, it outperforms even search instructions built into the computer hardware.

For computers that lack a search instruction, we advocate use of the simplified Boyer and Moore algorithm (algorithm SBM in this paper). For computers that do have such an instruction, the best approach appears to be the composite strategy summarized by:

```
if patlen <= threshold
    then search with slfc method
    else search with sbm method;
```

On the Amdahl V7 computer, we measured the value of THRESHOLD to be 5. It may, of course, be slightly different on other computers.

ACKNOWLEDGEMENT

This work was supported by a grant from the National Science and Engineering Research Council of Canada.

## REFERENCES

1. R. S. Boyer and J. S. Moore, 'A fast string searching algorithm', *CACM,* **20** (10), 762–772 (1977).
2. Burroughs Corporation, *B6700 Information Systems Reference Manual.*
3. Z. Galil, 'On improving the worst case running time of the Boyer-Moore string matching algorithm', *CACM,* **22** (9) 505–508, (1979).
4. IBM Corporation, *System/370 Principles of Operation.* Form No. GA22–7000.
5. D. E. Knuth, J. H. Morris Jr. and V. B. Pratt, 'Fast pattern matching in strings', *SIAM J. Computing,* **6** (2) 323–350, (1977).
6. Sperry Rand Corporation, *1100/80 Processor and Storage.* Publication No. UP 8492.