Edgar H. Sibley Panel Editor

Daniel M. Sunday

A VERY FAST SUBSTRING SEARCH ALGORITHM

This article describes a substring search algorithm that is faster than the Boyer-Moore algorithm. This algorithm does not depend on scanning the pattern string in any particular order. Three variations of the algorithm are given that use three different pattern scan orders. These include: (1) a "Quick Search" algorithm; (2) a "Maximal Shift" algorithm; and (3) an "Optimal Mismatch" algorithm.

A fundamental technique used in computer science is to search for a specific substring in a larger body of text. Algorithms that do this rank with sorting algorithms as cornerstones of software methodology. Substring search algorithms can be used to find reference keywords in documents and all usages of some variable in source code, to monitor input text streams for certain event names or prompt words, or to locate items in a list stored in a computer as flat text. The string search technique is so fundamental that most large computer programs use it in one form or another. Often the execution of this technique in code accounts for a substantial percentage of the work a program does, and increases in the efficiency of these search routines can significantly speed up a computer program.

The substring search problem is to find all occurrences of a given pattern string p as a substring of a larger string of text t. Several important algorithms have been discovered that are more efficient than the straightforward (SF) approach. Two of the most notable algorithms, published over a decade ago, are the Knuth-Morris-Pratt (KMP) [3] and the Boyer-Moore (BM) [1] algorithms. Both the KMP and BM algorithms have worst-case linear-search behavior, improving the quadratic SF algorithm. In practice, however, on commonplace English text, the BM algorithm is several (usually three or more) times faster than the other two that are about the same [4]. In this article, an improvement to the BM algorithm is presented that results in an even faster substring search algorithm. This new algorithm does not require that the pattern string be scanned in any particular order. Three variations of the algorithm are given that use three different pattern string scan orders. These include: (1) a "Quick Search" algorithm; (2) a "Maximal Shift" algorithm; and (3) an "Optimal Mismatch" algorithm. All three are very fast substring search algorithms.

Existing Algorithms

Let p[i] be the i-th character in the pattern string p=p[0]...p[$m-1$] of length m, and let t[j] be the j-th character in the text string t=t[0]... t[$n-1$] of length $n>m$. We will assume that the pattern string p is located at position k in the text string t in testing for a substring match. That is, p[0] is aligned with t[k], p[1] is aligned with t[$k+1$], and p[i] is aligned with t[$k+i$] up to $i=m-1$.

The SF algorithm is the obvious one that most programmers would use to code a substring search. The pattern string p is aligned with the extreme left of the text, at position $k=0$, and then the pattern characters are scanned from left to right, p[0] p[1]...p[$m-1$], testing for matches against the corresponding text characters. If all match, then a substring has been found. If any mismatch is found, the pattern string is shifted to the right one step, incrementing k by 1, and the pattern string is rescanned left to right starting again from its leftmost position at p[0]. This SF algorithm is easy to code. The main drawback of using the SF algorithm is that it is a quadratic algorithm with worst-case $O(mn)$ search time. In practice, however, for each pattern string position, a mismatch is usually detected with the first character tested, and the expected running time is $O(n)$.

The KMP algorithm [3] improves the SF algorithm with a worst-case $O(m+n)$ search time. The funda-

mental idea behind the KMP algorithm is to use already-known matches to permit shifting the pattern string forward by a delta, δ, of more than one character when a mismatch is found. The KMP method starts the same as the SF and scans the pattern string in the same left to right direction from p[0] through p[$m-1$]. When a character mismatch is found, however, between p[i] and t[$k+$i], for example, the KMP algorithm shifts the pattern string right in order to align already-matched and scanned text with the nearest matching prefix of the pattern. Additionally, a different pattern character is brought to the mismatch position, since we already know that the current one is a mismatch. These prefix shifts for each mismatch position in the pattern string can be determined from the pattern string alone, and an initial $O(m)$ time is needed to precompute them. After this shift of $\delta >= 1$, testing characters of the text string then resumes at the point where the last mismatch was found, namely, at t[$k+$i] in the text string and at p[$i-\delta$] in the pattern string when $(i-\delta) >= 0$. If $(i-\delta) < 0$, p is shifted to position $(k+i+1)$ in t. Thus, there is no backtracking in t, resulting in a worst-case $O(n)$ search time. Nevertheless, in practice, the KMP and SF algorithms perform about the same [4] because the expected search time statistics are dominated by the event of a mismatch for the first character tested, p[0].

The BM algorithm [1] changes the direction of scanning the pattern string by testing the last character of it first and then proceeding right to left through the pattern string, p[$m-1$] p[$m-2$]. . . p[0], in testing for matches with the text. Similar to KMP, when a mismatch is found, at p[$m-$i], for instance, the information gained from known matches is used to shift the pattern right as much as possible. The shifts are generally larger than those for the KMP algorithm. For the mismatching character in the text string, t[$k+m-i$], the BM algorithm uses a precomputed table to find the index of its first leftward occurrence from the end of the pattern string. If this occurs to the left of the position of the mismatch, then the difference is defined to be δ_1 for that position in the pattern string. Since most often, the last character of p, p[$m-1$], gives a mismatch, δ_1 is generally positive. For reasonably short patterns, the expected value of δ_1 is almost $(m-1)$. Using the BM δ_1 for the pattern shift after a mismatch yields a substring search algorithm that is usually better than three times as fast as those of the SF or KMP in practice [4]. This improved algorithm, however, has a worst-case $O(mn)$ search time. By further incorporating the KMP idea to compute a δ_2 and using the maximum of δ_1 and δ_2 for the actual δ shift used, one gets an $O(m+n)$ algorithm. This second shift value is computed by taking the already-matched suffix of p to the right of the first mismatched character and by finding the next leftward occurrence of it in p. Additionally, the character at the mismatch position must be different from the current one. Like δ_1, δ_2 is precomputed as a function of the position in p where a mismatch first occurs, and both δ_1 and δ_2 can be precomputed in $O(m)$ time.

An Improved Algorithm

One can recode the intent of the δ_1 of the BM technique. First, note that the pattern string always shifts right by at least one character. Hence, the character in the text string immediately past the end of the pattern string, namely t[$k+m$], must be involved in testing for a substring match at the next position of the pattern string. Thus, a new Δ_1 can be computed to be the index of the first leftward occurrence of this character from the end of the text string. As with the BM δ_1, this index can be precomputed as a function of the text string alphabet, for instance, in a table TD1[c], whose value for any character c of the alphabet is its leftward index from the end of p (so that the last character of p has index 1). Then,

$$\Delta_1 = \text{TD1}[\text{t}[k+m]].$$

Whenever a mismatch is found, this value of Δ_1 is the amount of shift p to the right. This either aligns that character in p with the text character t[$k+m$], or, when that character does not occur in p, shifts p right past it to text position $(k+m+1)$. Note that this Δ_1 is computed as an absolute pattern shift and is not defined relative to the position in p of the last mismatch. Using this Δ_1 instead of the BM δ_1 has the following advantages:

(1) $\Delta_1 >= 1$ always, and so it can be used by itself to simply and quickly code a fast, practical algorithm. The BM δ_1, however, is sometimes $<= 0$, in which case either a shift of just 1 or δ_2 is used.

(2) In practice, one expects that $\Delta_1 >= \delta_1 + 1$. Also, whenever the last character of the pattern string matches the text character, then one expects that $\Delta_1 >= \delta_1 + 2$, and so on. Thus using Δ_1 results in a faster algorithm than that of BM. This is mostly true for short pattern strings, and the effect of this increase in speed decreases as the pattern string gets longer.

(3) Δ_1 does NOT depend on the order in which the pattern string p is scanned. This is because it is defined relative to a text string character that lies outside the current comparison range of the pattern string. The BM δ_1, however, depends strongly on the right to left pattern string scan order for its definition and efficient usage.

This last point is important, for it means that *the pattern string p may be scanned in any order at all.* One could scan it forward, backward, or use any other ordering of the subindices of the pattern string. Let an index ordering be represented as an integer array I[ ] = {I[0], . . . , I[$m-1$]} that is a permutation of $\{0, \ldots, m-1\}$. Then, I[j] is the location in the pattern string of the j-th scan element, and p[I[j]] is the character of the pattern string at that location, for each $j = 0 \ldots (m-1)$.

For any specific order of scanning a pattern string, one can define a Δ_2 shift that is similar to the KMP δ or

Algorithim (1)

```
{ first initialize TD2[] for the minimum matching shift }
TD2[0]:= 1; { no match }
lshift:= 1;
for j:= 1 to (m - 1)
do begin { scan further leftward for first matching shift }
  lshift:= matchshift(j,lshift);
  TD2[j]:= lshift;
end;

{ next get correct shift with current char mismatch }
for j:= 0 to (m - 1)
do begin
  gotshift:= false;
  lshift:= TD2[j]; { get initial matching shift }
  while (gotshift = false) and (lshift < m)
  do begin { already have a matching shift }
    { also require current char must not match }
    i:= (I[j] - lshift);
    if (i < 0) or (p[I[j]] <> p[i])
    then gotshift:=true
    else begin { get next matching shift }
      lshift:= lshift + 1;
      lshift:= matchshift(j,lshift);
    end;
  end;
  TD2[j]:= lshift; { set final shift }
end;
```

Algorithim (2)

```
{Search for a pattern in text}
gotmatch := false;
k := 0;
while (gotmatch = false) and (k + m <= n) {enough text is still left}
do begin
  j:=0; {j scans the ordered pattern }
  while (j < m) and (p[ I[ j ] ] = text[ k + I[ j ] ])
    do j := j + 1;
  if (j = m) {all pattern chars matched}
  then gotmatch := true
  else begin { shift pattern }
    delta1 := TD1 [ text[ k + m ] ];
    delta2 := TD2[ j ];
    k := k + max ( delta1, delta2 );
  end;
end;
if (gotmatch = true)
then Search:= k        { pattern match found at text location k }
else Search := (-1)    { no pattern match found in text}
```

the BM δ_2. Then, for the substring search algorithm, one uses a Δ pattern shift that is the maximum of Δ_1 and Δ_2. Like the BM δ_2, this new Δ_2 can be precomputed as a function of the position in p where a mismatch first occurs. Let TD2[j] be the precomputed Δ_2 for when a mismatch first occurs at I[j] for the scan ordering I[ ]. To precompute the table TD2[ ], first consider p[I[0]]. If a mismatch occurs testing this character, we know that the next character of p that becomes aligned with the corresponding text location must differ from p[I[0]]; otherwise there would be another mismatch. Find the maximum i<I[0] such that p[i] does not equal p[I[0]]. Then, TD2[0]=(I[0]−i) is the minimum shift where this holds. Next, TD2[1] is the minimum amount one must shift p left so that p[I[0]] matches its corresponding character, but p[I[1]] does not.

Continue defining TD2[j] to be the minimum left shift so that p[I[0]] ... p[I[j-1]] match their aligned characters in p, but such that p[I[j]] does not. The Δ_2 shift table for a specific ordered pattern can be precomputed with Algorithm (1) (Note: the algorithms described in this article will be given in Pascal for clarity of presentation. A complete implementation in the C language is also given in the Appendix.) where the matchshift (j, lshift) function returns the value of the next leftward shift, after an initial left shift **lshift**, for which each of the first j-ordered pattern characters match their corresponding aligned string character. That is, this is the minimum value of **mshift**>= lshift>=0, such that either (I[i]− mshift)<0, or p[I[i]]=p[I[i]−mshift], for each i=0...(j−1).

Note that if the pattern string p is scanned in the forward direction, with I[ ]={0, 1, ..., m−1}, then the Δ_2 we have computed is the KMP δ. Also, if the pattern string is scanned in the reverse direction, with I[ ]={m−1, m−2, ..., 0}, our Δ_2 is the same as the BM δ_2.

Given a specific ordered pattern and precomputed shift tables TD1[ ] and TD2[ ] for it, the new substring search algorithm (Algorithm (2)) is easy to code.

Similar to the KMP and BM algorithms, Algorithm (2) should have linear O(n) worst-case behavior for any scan order. It would seem that the KMP algorithm uses a best possible scan order for remembering already scanned text in order to avoid backtracking and should produce the best worst-case behavior. On the other hand, the BM scan order produces the worst possible performance, since it does not remember any scanned text. A proof of the O(n) linearity of the search algorithm for any fixed scan order of the pattern string might include the KMP and BM algorithms as special cases and reveal a relation between the bounds on their worst-case behavior. The proof of the linearity of this new search should be similar to the proof for the BM algorithm [2, 3]. The details of a complete proof, however, have not yet been worked out, so we simply conjecture that linearity holds.

- *Conjecture:* The arbitrary scan order substring search algorithm has O(n) worst-case behavior.

Having a Δ_1 and a Δ_2 that can be used with any substring scan order creates new algorithmic possibilities. Three different variations based on three different scan orderings are given here.

The Quick Search (QS) Algorithm.

To quickly code a fast substring search algorithm, the easy-to-code SF pattern string scan order can be used with the easy-to-compute Δ_1 for the pattern string shift at each stage. No Δ_2 is used. This is a simple, fast practical algorithm. Because it can be both coded and debugged quickly and it executes quickly, it can be called the *Quick Search* algorithm. This simplified search algorithm is shown in Algorithm (3).

If one augmented this straightforward search with the Δ_2 shift and added code to stop backtracking in the text, one would get a fast algorithm with the KMP algorithm tight bound on worst-case behavior.

The Maximal Shift (MS) Algorithm.

One can try to choose a scan order that somehow maximizes the Δ_2 shift values that depend on it. One way of doing this is to first pick the character in the pattern string p whose next leftward occurrence in p is a maximal distance away. Test this character

Algorithm (3)

```
{Quick Search for a string in text}
gotmatch := false;
k := 0;
while (gotmatch = false) and (k + m <= n)
do begin
  i := 0; { i scans the pattern string }
  while (i < m) and (p[ i ] = text [ k + i ])
    do i := i + 1;
  if (i = m) {all pattern chars matched}
  then gotmatch := true
  else k := k + TD1[ text[ k + m ] ] { shift pattern }
end;
if (gotmatch = true)
then QSearch := k      { substring match found at text[k]}
else QSearch := (-1)  { no substring match found in text}
```

first. If it matches the corresponding text character, then we have to maximally shift the pattern string p right, before the next valid comparison position is reached. Repeat this selection process with the remaining characters of p. One could also take into account the character where a mismatch is first detected to skip over pattern string subsequences of repeated characters. Other refinements could maximize higher-order Δ_2 shifts, but this would not result in a significant increase of search efficiency.

C code for constructing a "Maximal Shift" (MS) ordered pattern is given in the Appendix. This code sorts with a comparison of the minimum left shifts needed to match the characters being compared. The sort first picks the character with the maximal minimum left shift. If two characters have the same minimum left-shift value, we use the BM heuristic to first select the one closest to the end of the pattern string.

The Optimal Mismatch (OM) Algorithm.

An algorithm that is even faster in practice can be achieved by using a pattern string scan order that optimizes the chance of getting a mismatch at each test position. This is done by ordering the characters of the pattern string p from the one least likely to occur in the text alphabet to the one most likely to occur. Use this as the pattern scan order. This increases the probability of finding a mismatch as soon as possible and results in greater efficiency.

C code for constructing an "Optimal Mismatch" (OM) ordered pattern is given in the Appendix. This code sorts with a comparison of the frequency of occurrence of the pattern string's characters in the text alphabet. When two characters have the same frequency, the BM heuristic is used to first select the one closest to the end of the pattern string. One could go beyond this by first selecting, for characters of near equal frequency, the one that would give the maximal Δ_2 shift. Even further, one could compute the number of expected comparisons for each character that would result from the probability of it matching or not matching and the expected shift associated with each event. This does not give much better results than the simplified OM algorithm that we have used.

TABLE I. *Fraction of Text Characters Tested*

Plen	Words	BM	QS	MS	OM
1	26	1.000	0.529	0.529	0.529
2	131	0.543	0.390	0.390	0.377
3	741	0.376	0.299	0.300	0.288
4	2142	0.293	0.246	0.251	0.237
5	3077	0.242	0.213	0.216	0.204
6	3773	0.210	0.190	0.194	0.181
7	3911	0.186	0.171	0.174	0.163
8	3474	0.168	0.157	0.159	0.150
9	2965	0.155	0.147	0.148	0.140
10	1881	0.144	0.138	0.139	0.131
11	1051	0.137	0.132	0.132	0.125
12	542	0.129	0.125	0.124	0.118
13	260	0.124	0.120	0.119	0.114
14	102	0.117	0.115	0.113	0.109
15	39	0.115	0.113	0.109	0.106

To illustrate the impact of the OM algorithm, note that over 20 percent of English words end in the letter 'e', the most commonly occurring character in English text with about a 10 percent occurrence rate. Thus, many words that are searched for in text using the BM algorithm often get a match for the first character tested. Testing the least probable character of a word first, considerably improves this statistic. The average ratio of the text occurrence probability of the last letter of a word to the least likely letter in it is almost 5, making a mismatch on the first character tested five times more probable in general. For some words (one percent), this ratio can be as high as 50 or more. For words ending in 'e', the average ratio is almost 9.

Comparison of the Algorithms.

To compare the BM, QS, MS, and OM algorithms, each was used to search for the same strings in large fixed text buffers, and the number of comparisons made with text characters was counted. The algorithms were coded in the C programming language (see Appendix). The first text buffer used was formed from the UNIX™ spelling dictionary file, /usr/dict/words, by discarding non-alphabetic characters and converting alphabetics to lower case. This resulted in about 200K characters of text. Then, all occurrences of each alphabetic word in /usr/dict/words were searched for in this text buffer using each of the four algorithms. After counting the number of character comparisons made for each word and algorithm, the fraction of the total characters of text was computed and recorded. Finally, for each pattern string length, the average of this fraction was computed for each algorithm. The resulting statistics (Table I) show that the QS, MS, and OM algorithms are all faster than the BM algorithm. Table I compares the BM, QS, MS, and OM algorithms as a function of the pattern string length, "Plen." The "Words" column gives the number of words of each length searched for in the text. These results show that the OM algorithm is the fastest one of all.

Next, the increase in speed of the fastest algorithm, the OM algorithm, over the BM algorithm, was computed as the ratio BM/OM of the values in Table I for each word, and the average for each pattern length was computed as shown in Table II. This shows a dramatic increase in

UNIX is a trademark of AT&T Bell Laboratories

TABLE II. *OM Compared to BM*

Plen	BM/OM	Min	Max
1	1.89	1.71	1.99
2	1.45	1.35	1.53
3	1.30	1.18	1.41
4	1.23	1.10	1.34
5	1.19	1.07	1.30
6	1.16	1.05	1.27
7	1.14	1.02	1.25
8	1.12	1.02	1.23
9	1.11	1.01	1.20
10	1.10	1.01	1.19
11	1.09	1.01	1.18
12	1.09	1.00	1.17
13	1.08	0.99	1.16
14	1.08	1.00	1.14
15	1.08	1.02	1.16

TABLE III. *Manual Page Text Comparison of BM and OM*

Plen	BM	OM	BM/OM	Min	Max
1	1.000	0.522	1.92	1.74	2.00
2	0.533	0.367	1.46	1.35	1.52
3	0.365	0.278	1.31	1.18	1.40
4	0.282	0.228	1.24	1.11	1.32
5	0.232	0.194	1.19	1.04	1.33
6	0.200	0.171	1.16	1.05	1.32
7	0.175	0.153	1.14	1.01	1.29
8	0.157	0.139	1.13	1.02	1.23
9	0.144	0.128	1.12	1.02	1.20
10	0.132	0.119	1.11	1.02	1.19
11	0.124	0.112	1.10	1.00	1.19
12	0.115	0.105	1.09	1.02	1.17
13	0.109	0.100	1.09	1.02	1.17
14	0.103	0.095	1.08	1.02	1.13
15	0.099	0.091	1.08	1.02	1.17

search speed for short pattern strings and a general increase of almost 10 percent for longer strings. Table II also shows the minimum and maximum values of the BM/OM ratio for any individual word that occurred for each string length. This shows that the OM algorithm is at least as good as the BM one and can sometimes be significantly faster.

A further test to compare the BM and OM algorithms was done using the text buffer formed by concatenating all the UNIX manual pages. The raw, unformatted manual pages were only filtered by throwing away format command lines and by converting alphabetic characters from upper to lower case. Nonalphabetic special characters, including all white spaces, were retained in the text. This resulted in almost 3 megabytes of technical English text. Again, each alphabetic word from the UNIX dictionary was searched for in this text buffer using the BM and OM search algorithms. The results are given in Table III. It is interesting to note that the speedup ratio BM/OM, as a function of Plen is almost exactly the same as computed in the dictionary text search tests.

Conclusion

Throughout the history of computer science, there has been an evolving discovery of new, fast string search algorithms. Theoretical work in automata theory, in the 1960s, led directly to the algorithms of the 1970s. Of these, the Boyer-Moore (BM) algorithm became notorious as the fastest technique available to search for a single fixed substring. It was also notorious for going against natural intuition by scanning the pattern string in reverse order and thus gaining its startling efficiency. Following this work, many improvements have been made to the BM algorithm that can increase search speed for certain types of patterns. All of these improved algorithms, however, still depend on the BM technique of scanning the pattern string in reverse order to achieve their efficiency.

This article has presented an extension of the BM algorithm that does away with dependence on the scan order of the pattern string. In fact, the pattern can be scanned in any arbitrary order, and there is still an increase in efficiency over the BM algorithm. It is then shown how to select scan orders that increase this efficiency even more. Three specific new algorithms are presented that use three different pattern string scan orders. These algorithms are called the "Quick Search" (QS), the "Maximal Shift" (MS), and the "Optimal Mismatch" (OM) algorithms.

The first of these, the QS algorithm, is very easy to implement and scans the pattern in the most natural forward order. It is almost as easy to understand, code, and debug as the slow, straightforward algorithm that most programmers tend to use. Using the QS algorithm, however, will most often give superior search speeds to even the BM algorithm. When a programmer is called on to rapidly code a string search, the QS algorithm should be his or her choice.

The final algorithm, the OM algorithm, is the fastest one of all. It gains its efficiency by first testing the least probable pattern string character and thus detects mismatches more quickly. This event dominates search statistics and results in a significant increase in speed (see Tables I, II, and III). The greatest gains are for short pattern strings, where there is a 20 percent or greater increase in search speed for normal English text. For longer strings, the relative advantage becomes smaller, and one can expect a text search speed increase of about 10 percent.

In applications not involving English text, the algorithms presented in this article should still give better performance than the Boyer-Moore string search. The statistics on the degree of improvement would be different since they depend on the size of the text alphabet and the frequency of occurrence of the alphabet characters in the application text.

Acknowledgments.
Although this work was not the result of a directly funded research project, it is most certainly a side-effect of

the projects I have worked on at JHU/APL. I would like to acknowledge the stimulating technical environment in which I work and the resources made available to me on a daily basis. These factors have made this work possible.

References

1. Boyer, R.S., and Moore, J.S. A fast string searching algorithm. *Commun. ACM 20,* 10 (Oct. 1977), 762-772.
2. Guibas, L.J., and Odlyzko, A.M. A new proof of the linearity of the Boyer-Moore String Searching Algorithm. *SIAM J. Comput. 9,* 4 (Nov. 1980), 672-682.
3. Knuth, D.E., Morris, J.H., and Pratt, V.R. Fast pattern matching in strings. *Siam J. Comput. 6,* 2 (June 1977), 323-350.
4. Smit, G.V. A comparison of three string matching algorithms. *Softw.—Prac. and Exp. 12,* 1 (Jan. 1982), 57-66.

CR Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sorting and searching*

General Terms: Algorithms

Additional Key Words and Phrases: Boyer-Moore, Knuth-Morris-Pratt, pattern, search, *substring*

About The Author:

DANIEL M. SUNDAY is currently a mathematician at the Johns Hopkins University Applied Physics Laboratory (JHU/APL) and works as the designer and developer of software systems. He is also the Lead Software Engineer on a project developing color display systems for the U.S. Navy. Author's Present Address: The Johns Hopkins University, Applied Physics Laboratory, Johns Hopkins Road, Laurel, MD 20723-6099. dan@aplvax.jhuapl.edu

APPENDIX
C-Language Implementation

In this appendix, the algorithms presented in this article are given in the C programming language. This is the code that was used to test the algorithms and is a complete and correct implementation of them. The code is also a reasonably efficient C representation of the algorithms, although readability of the code took precedence over raw speed.

In the C code, the global variables `Plen` and `Tlen` are assumed to be preset to the pattern and text string lengths. That is, `Plen`$=m$, and `Tlen`$=n$. Also, the constant `ASIZE` is the size of the text alphabet that is assumed to be the ASCII character set. The constant `MAXPAT` is the maximum length of a pattern string. The pattern string is represented as a NULL-terminated string declared as an array:

```
char pstr[ MAXPAT ];
```

Given a specific pattern scan order `I[]`$=\{$`I[0]`, ..., `I[`$m-1$`]`$\}$, the ordered pattern is represented by an array of structures

```
typedef struct pattern_scan_element {
        int     loc;            /* location in pstr of scan element */
        char    c;              /* character of pstr at scan location */
        }
        PAT;

PAT     pattern[ MAXPAT ];      /* a specific ordered pattern */
```

where `pattern[`j`] . loc=I[`j`], pattern[`j`]. c=pstr[ I[`j`] ]` for j=0...(m–1), and `pattern [`m`] . c=0`.

Often an ordered pattern can be constructed from a pattern string by sorting with a comparison function `pcmp()` that compares two `PAT` elements according to the specification for the UNIX `qsort()` function. The following function does this.

```
/* order_pattern(): construct an ordered pattern from a string
*/
order_pattern( pstr, pcmp, pattern)
char *pstr;             /* input: the pattern string */
int  (*pcmp)();         /* input: routine to compare PAT elements */
PAT  *pattern;          /* output: the scan ordered pattern */
{
     int i;
     PAT *pat=pattern;

     for (i=0; i<=Plen; ++i, ++pstr, ++pat) {
         pat->loc = i;
         pat->c = *pstr;
     }
     qsort( pattern, Plen, sizeof(PAT), pcmp);
}
```

For example, the comparison function for the Maximal Shift pattern ordering is:

```
maxshift_pcmp( pat1, pat2 ) /* "Maximal Shift" pattern comparison */
PAT *pat1, *pat2;  /* input: pointers to two PAT elements */
{
   int    dsh = MinShift[ pat2->loc ] - MinShift[ pat1->loc ];

   return (dsh ? dsh : (pat2->loc - pat1->loc));
}
```

where the array `MinShift[ j ]` gives the minimum left shift needed to match the pattern string character at location j. Note that if two characters have the same `MinShift` value, we use the BM heuristic to first select the one closest to the end of the pattern string. The `MinShift[]` array is easily computed with the following code.

```
for (i=0; i<Plen; ++i) {
       for (j=i-1; j>=0; --j)
            if (pstr[j]==pstr[i]) break;
       MinShift[ i ]=(i - j);
}
```

Also, the comparison function for constructing the Optimal Mismatch pattern ordering is:

```
optimal_pcmp( pat1, pat2 ) /* "Optimal Mismatch" pattern comparison */
PAT *pat1, *pat2;  /* input: pointers to two PAT elements */
{
    float  fx = Freq[ pat1->c ] - Freq[ pat2->c ];

    return (fx ? (fx>0 ? 1 : -1) : (pat2->loc - pat1->loc));
}
```

where the array `Freq[c]` returns the frequency of occurrence of the character `c` in the text alphabet. The values used in our tests with English text for the percentage frequency of occurrence of alphabetic characters is given in Table I. This table was derived from the UNIX spelling dictionary. Most other nonalphabetic characters have low-occurrence frequencies. Note, however, the most common character in English text, the space character, with about 15 percent occurrence, is not included. This is relevant in applications where the pattern string can have embedded blanks.

TABLE I. *English Text Alphabet Frequency*

Char	Freq	Char	Freq
e	11.1	p	3.1
a	8.9	h	2.9
i	7.8	g	2.4
r	7.4	b	2.3
t	7.1	y	2.0
o	6.9	f	1.5
n	6.8	w	1.1
s	5.6	k	1.1
l	5.5	v	1.0
c	4.5	x	0.3
u	3.6	j	0.2
m	3.2	z	0.2
d	3.2	q	0.2

Given a pattern string and the associated scan-ordered pattern, the functions to precompute the Δ_1 and Δ_2 shift tables `TD1[]` and `TD2[]` are the following:

```
/* build_TD1(): constructs the delta 1 shift table from a pattern string
*/
int    TD1[ ASIZE ];                    /* output: table for delta 1 shift index */

build_TD1( pstr )
char   *pstr;                           /* input: the pattern string */
{
       int    i;
       char   *p;

       for (i=0; i<ASIZE; i++)          /* initialize the TD1[] table */
              TD1[i] = Plen + 1;
       for (p=pstr; *p; p++)            /* fill in values from pattern string */
              TD1[ *p ] = Plen - (p - pstr);
}

/* build_TD2(): constructs the delta 2 shift table from an ordered pattern
*/
int    TD2[ MAXPAT ];                   /* output: table for delta 2 shift index */

build_TD2( pstr, pattern )
char   *pstr;                           /* input: the actual pattern string */
PAT    *pattern;                        /* input: the scan-ordered pattern */
{
       int    lshift;                          /* current left shift */
       int    i, ploc;                         /* pattern location counters */

       /* first initialize TD2[] for the minimum matching left shift */
       TD2[ 0 ]=lshift=1;                       /* no preceding chars, so=1 */
       for (ploc=1; ploc<Plen; ++ploc) {        /* for each pattern location */
```

```
            /* scan leftward for first matching shift */
            lshift = matchshift( pstr, pattern, ploc, lshift );
            TD2[ ploc ] = lshift;                    /* set initial matching shift */
        }
        /* next get correct shift with current char mismatch */
        for (ploc = 0; ploc<Plen; ++ploc) {
            lshift = TD2[ ploc ];                    /* get initial matching shift */
            while (lshift<Plen) { /* when current shift is less than pattern len */
                /* already have a matching shift here */
                /* also require current char must not match */
                i = (pattern[ploc].loc - lshift);
                if (i<0 || pattern[ploc].c != pstr[i])      /* mismatch */
                  break;
                /* if not, scan further for next matching shift */
                ++lshift;
                lshift = matchshift( pstr, pattern, ploc, lshift );
            }
            TD2[ ploc ] = lshift;                              /* set final shift */
        }
}

    /* matchshift(): find the next leftward matching shift for the first
    **    ploc pattern elements after a current shift of lshift.
    ** output: return this next left shift value.
    */
    matchshift( pstr, pattern, ploc, lshift )
    char *pstr;          /* input: the pattern string */
    PAT  *pattern;       /* input: the ordered pattern */
    int  ploc;           /* input: the number of pattern elements to match */
    int  lshift;         /* input: the smallest left shift to consider */
    {
        PAT      *pat;
        int      j;

        for ( ; lshift<Plen; ++lshift) { /* scan left for matching shift */
                pat = pattern + ploc; /* current pattern element */
                while (--pat>=pattern) {
                    /* all preceding chars must match */
                    if ((j = (pat->loc - lshift))<0)
                      continue;
                    if (pat->c != pstr[ j ]) break;
                }
                if (pat<pattern) break;/* all matched */
        }
        return lshift;
    }
```

After `TD1[]` and `TD2[]` have been precomputed for the scan-ordered pattern, the search algorithm is given by the following function. This function returns the index of the first instance of the pattern in the text. If no matching substring is found, it returns a `(-1)`.

```
    /* search(): the arbitrary scan order substring search algorithm.
    ** output:   return the text index of the substring, or (-1) if none.
```

```
*/
search( pattern, text )
PAT   *pattern;            /* input: a scan-ordered pattern string */
char *text;                /* input: the text */
{
     PAT   *p;                /* pattern scan pointer */
     char *tx=text;           /* text scan pointer */
     int  d1, d2;             /* deltas for pattern shift */

     while (tx+Plen <= text+Tlen)/* while enough text is still left */
     {
          for (p = pattern; p->c; ++p) {        /* scan the pattern*/
                  if (p->c != *(tx + p->loc))/* got a mismatch */
                    break;                      /* so stop checking */
          }
          if (p->c == 0)                 /* pat end=>got substring */
                  return (tx - text);    /* return index into text */

          /* no substring match, so shift to next text position */
          d1 = TD1[ *(tx + Plen) ];     /* get delta1 */
          d2 = TD2[ p - pattern ];      /* get delta2 */
          tx += (d1>d2 ? d1 : d2);      /* use max for shift */
     }
     return (-1); /* no substring found */
}
```

The simplified Quick Search function, which only uses Δ_1 and only needs to have `TD1[]` precomputed, is given by the following function:

```
/* qsearch():        the quick substring search algorithm.
** output:          return the text index of the substring, or (-1) if none.
*/
qsearch( pstr, text )
char *pstr;             /* input: the pattern string */
char *text;             /* input: the text */
{
     char *p;              /* pattern string pointer */
     char *t, *tx=text;    /* text pointers */

     while (tx + Plen <= text + Tlen)   /* while enough text is still left */
     {
          for (p=pstr, t=tx; *p; ++p, ++t) {        /* scan pattern string */
              if (*p != *t) break;                  /* mismatch, so stop */
          }
          if (*p == 0) return (tx - text); /* got substring, return index */
          /* no substring match, so shift to next text location */
          tx += TD1[*(tx + Plen)];                      /* shift by delta1 */
     }
     return (-1);                  /* no substring found */
}
```