SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(11), 1221–1248 (NOVEMBER 1991)

Fast String Searching

ANDREW HUME
AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974, U.S.A.

AND

DANIEL SUNDAY
Johns Hopkins University / Applied Physics Laboratory, Johns Hopkins Rd., Laurel, MD 20723, U.S.A.

SUMMARY

Since the Boyer-Moore algorithm was described in 1977, it has been the standard benchmark for the practical string search literature. Yet this yardstick compares badly with current practice. We describe two algorithms that perform 47% fewer comparisons and are about 4.5 times faster across a wide range of architectures and compilers.

These new variants are members of a family of algorithms based on the skip loop structure of the preferred, but often neglected, fast form of Boyer-Moore. We present a taxonomy for this family, and describe a toolkit of components that can be used to design an algorithm most appropriate for a given set of requirements.

KEY WORDS String searching Pattern matching Boyer-Moore

1. INTRODUCTION

Searching for a string in a body of text is a fundamental concern of computer science and applications. Yet, partially because the best algorithms presented in the literature are difficult to understand and to implement, knowledge of fast and practical algorithms is not commonplace. In this paper we present a taxonomy and an organizational framework for categorizing and constructing string search algorithms, and show how the best known algorithms are classified within it. We then use this taxonomy to examine the options for the various components of a search algorithm, and compare their execution speeds on a variety of architectures for a natural language test set. As a result, we develop new algorithms that execute more rapidly than those previously known.

Theoretical work on string searching has focused on the worst case computational complexity and the asymptotic search behavior for long strings by counting the number of character comparisons made. For the problem of searching for all occurrences of a string of m characters in a text of n characters, we denote the number of text comparisons made by an algorithm in the worst case as $c(n, m)$. An upper bound of $c(n, m) \geq 2n - m + 1$ was achieved by the Knuth-Morris-Pratt (KMP) algorithm [1] in 1977. The worst case perfor-

0038–0644/91/111221–28$14.00

Received 4 January 1991
Revised 3 May 1991

mance of the Boyer-Moore (BM) algorithm,[2] despite its astonishing speed in practice, was initially shown to be $c(n,m) \geq 6n$.[1] This has been improved over the years, finally reaching a tight bound of $c(n, m) \geq (3n - n/m)$ by Cole.[3] A variant of Boyer-Moore designed by Apostolic and Giancarlo[4] achieved a bound of $C(n, m) \geq 2n - m + 1$. Recently, Colussi, Galil and Giancarlo[5] described an algorithm based on the KMP and BM algorithms where $7n/6 \geq c(n,m) \geq (4n - m)/3$.

More practical investigations of string searching, such as Horspoo1[6] and Smit,[7] evaluate algorithms by their run time as well as character comparisons, and investigate improving performance for typical natural language text searches, rather than pathological theoretical examples. Run time is arguably the most important metric but it is hard to measure reliably across different applications, implementations, and environments. In these practical studies, BM is the dominant algorithm.

In fact, Boyer and Moore described two algorithms.[2] The first, which we will call *classic BM*, was used for presentation and analysis of the algorithm and became the standard reference for comparison in subsequent literature. The second, which we call *fast BM*, was described as the form preferred for implementation because of its superior run time performance. Despite this, fast BM has largely been neglected in the literature in favor of the more easily analyzed classic BM (notable exceptions are HorspooI,[6] Woods[8] and Hume[9]). Below, we describe two descendants of the fast BM algorithm: a compact portable 'tuned' BM, and a slightly faster but more complicated 'least cost' search. Both are much faster than classic BM and set a new standard of performance for other algorithms to beat.

These new algorithms are just two members of a large family of algorithms with a common structure suggested by fast BM. After describing some user requirements that significantly affect the selection and design of an algorithm for string searching, we present our taxonomy for classifying string search algorithms and constructing them from a toolkit of algorithm fragments.

2. PROPERTIES OF STRING SEARCH ALGORITHMS

For the rest of this paper, we define the string searching problem as looking for all occurrences of a string *pat* of length *patlen* in another string *text* of length *textlen*. For the algorithms discussed, the related problem of finding the first match is a trivial adaptation. We present algorithms in the C programming language.[10] Keep in mind that C arrays have origin 0 indexing, and so the first character of the pattern is *pat* [0]. Our computational model is that *text is* in randomly accessible memory. The characters in *text* are accessed generally in increasing order but many of the algorithms require the ability to backtrack up to *patlen* characters. Some applications require an algorithm that processes *text* in monotonic increasing order; the only such algorithm we consider is KMP.

There are several plausible metrics for measuring the speed of an algorithm. The primary metric used in this paper is run time, or rather, the speed (in megabytes of text searched per second) for processing a specific test set described below. Another cost metric is the number of character comparisons made with the text; but this rarely matters in practice unless character comparisons are unusually expensive, such as with variable length encodings of characters or if the characters are compressed or encrypted. Nevertheless, comparison counts give a measure that is largely independent of language (and compiler) and the host architecture. These factors make the run time metric difficult to describe analytically and experiments are needed to evaluate it empirically. Another metric is the upper bound on the search time in the worst case. For example, a common implementation of fast BM has an average run time of $O(textlen/patlen)$ but a worst case performance of $O(textlen \times patlen)$.

The worst cases involve periodicities in the text and search strings. These rarely occur in the typical case of searching for words or variable names in English text or program source, but are more common in some applications such as searching genetic databases where the patterns often have cyclic subpattems. For these latter cases, and in applications with hard real-time limits on search times, one might choose an algorithm with a good worst case *0* (*textlen)* run time even though it may run slower on the average. Another aspect of an algorithm's performance is the time needed for preprocessing *pat* before really starting to search *text.* Although typically this time is *0* (parlen + | *alphabet* | *)* and thus negligible, it may dominate if *textlen* is small or if one is only looking for the first match and it is expected to be very close to the start of *text.* We do not consider preprocessing costs in this paper.

Much of the string search literature assumes nothing about the text to be searched (some exceptions are Horspool, [6] Kowalski and Meltzer, [11] Baeza-Yates, [12] and Sunday [13]). This may seem appropriate for a general string searching application such as the UNIX ® system tool *grep,* but in practice, most text searched either resembles a natural language (such as English) or a programming language. In these domains, certain statistical properties of typical text, such as the character frequencies, are sufficiently stable that algorithms can use a fixed distribution for them. We develop some algorithms that use character frequency information. We also measure the stability of their run times for the cases where the assumed distribution does not match that of *text.*

3. TESTING METHODOLOGY

The algorithms described in this paper had their performance measured by the following methodology. The various algorithms were implemented in C using normal efficient programming techniques, for example, using register variables and character pointers instead of array indices. The test harness read the text to be searched and all the search words into memory before timing started. The text was then searched for each word sequentially. To gauge the dependence of the algorithms on the system type, the tests were run on a variety of systems listed below. The code was compiled without change using the compiler options shown below. Note that the systems were chosen as a diverse set of conveniently accessible machines representing most modem architectures, and not as representative of all existing systems. We deliberately used a variety of compilers, rather than using a common compiler such as GNU's *gcc,* to demonstrate that a component's relative performance was mostly independent of the implementation system (hardware, compiler and libraries).

system	compiler	description
`386`	`cc -0`	AT&T (Intel 80386)
`68k`	`lcc`	AT&T Gnot (Motorola 68020)
`tray`	`cc -0`	Cray XMP128
`mips`	`cc -03`	SGI (MIPS R3000)
`spare`	`cc -04`	Sun SparcStation 2 (SPARC)
`vax`	`cc -0`	VAX 8550

The test results show the speed in MB of text searched per CPU second for each system, the average stride (step) of the skip loop through the input text and the percentage of input text characters accessed. The latter includes the accesses to step past mismatches (jump) as well as actual character comparisons between the pattern and the input text (crop). All the tests used the same input text — a randomly selected 1 MB subset of words from the King James Bible. Except for `cray,` the timings were all done on single-user systems with no

other processes running. For each algorithm, the mean timing of three runs was used, and the spread for these runs was recorded. The mean and standard deviation for the 37 recorded spreads are

	386	spare	mips	vax	68k	cray
spread	.35%	.48%	.40%	.39%	.15%	.15%
SD	.15%	.20%	,17%	.17%	.07%	.06%

The standard test was to look for all occurrences of 500 unique words, 200 selected randomly from the unique words in the whole bible and 300 selected from the lMB test subset. Of the 500 words, 428 were found in the test text, with 15228 matches in all. Word lengths varied from 2 to 16, the mean length was 6.95, and the standard deviation was 2.17. The other timing tests, for words of the same fixed length, used groups of 200 words or so selected randomly in the same fashion. We used the bible subset as the text to be searched because it is more representative of natural English text than the other convenient word lists (like dictionaries or on-line manual pages) and could be publicly released.

4. A TAXONOMY

The algorithms described here have a common structure: a repeated loop through a sequence of positions in the *text* being searched, and testing whether the *pattern* does or does not match the *text* at each position.

```
for i over text do
begin
    skip loop
    match (text [ i ] with pat )      comment fail at text [ i+p ] #pat [ p ]
    i := i + shift(i,p)
end
```

Starting at a given position in the *text*, there is first a *skip loop* (Boyer and Moore referred to this as the 'fast loop') which quickly skips past immediate mismatches in the *text.* When a possible match position is found, a *match algorithm* compares *pat* against *text* at that position. Finally, independent of whether a match is found, a *shift function* of the indices i and *p* where *text* [*i* +*p*] *#pat* [*p*] in the match test, increments the current search point in *text.* The shift is always positive and the algorithm progresses until it reaches the end of the *text.* We consider the following kinds of *skip loop* in detail in section 4.1.

Skip Loop Components	
none	the *skip loop* is omitted from the algorithm
sfc	search for first (leftmost) character in *pat*
sfc	search for least frequent character in *pat*
fast	search for last (rightmost) character in *pat* (fast BM)
ufast	portable, unrolled variant *of fast*
lc	least cost frequency dependent variant of *ufast*

The *match* component is an algorithm comparing *pat* with *text.* This algorithm has to test each character in the *pat* against its corresponding *text* character in some specific order. The order of comparison is significant; and some scan orders may have hardware or system support. Additionally, there may be a special *guard* test before the full match algorithm. The guard is simply a test for a match with a specific (low frequency) character before starting the full match, and can occur with any match scan order. We indicate the presence of a

guard test by adding a ‘ + *g* ’ suffix to the component name. Since the *om* scan already incorporates the guard notion, it has no guard version. We consider the following *match* algorithms in detail in section 4.2.

Match Algorithm Components	
fwd	forward linear scan (left to right)
rev	reverse linear scan (right to left)
om	optimal mismatch ascending frequency order
ms	order to maximize *sd2* shift component
fwd+g	test *guard* before *fwd* scan
rev+g	test *guard* before *rev* scan
ms+g	test *guard* before *ms* scan

The final component, the *shift* function, cannot be arbitrarily mixed with other components. For example, using *d2* below depends on (*i +p)* being the rightmost mismatch; thus the *rev* match must be used. We consider the following *shift* functions in detail in section 4.3.

Shift Function Components	
inc	1
kmp	KMP’s *next(p)*
d1	*BM’s delta* $_1$*(text* [*i+p*] *)*
d2	*BM’s delta* $_2$*(p)*
d12	BM’s *max(delta* $_1$*(text* [*i+p*] *), delta* $_2$*(p))*
sd1	Sunday’s *DELTA , (text* [*i +patlen*] *)*
sd2	Sunday’s *DELTA* $_2$*(p)*
md2	mini *sd2* on skip character
gd2	Giancarlo’s generalized *delta* $_2$*(p)*
multiple	use a combination of several other shifts

One easy way to combine multiple shifts is to take their maximum, which we denote by joining them with the ^ operator; for example, *d12* can be written *d* 1 ^*d2*.

An example might make our schema clearer. First, we note that while lower case italics are used for the names of components, full algorithms built out of components have capitalized names. The classic BM algorithm, denoted as BM.ORIG and shown below, is described by the triple (skip = *none*, match = *rev*, shift = *d12* = *d* 1 ^ *d* 2), or more tersely by { *none* | *rev* | *d* 1 ^ *d* 2].

```
i = patlen-1;
while (i < textlen) {     /* i scans thru the text */
     /* no skip loop */
     /' reverse match test */
     for (p = patlen-1;  (p >= 0) && (pat [p] == text [i]); p--)
          i--;
     if(p  <  0){     /* match at text [i+1] */
     }
     /* dl^d2 shift */
     i += max(delta1 [text [i] ] , delta2 [p] ) ;
}
```

Some more sample classifications are drawn from the literature:

Algorithm	skip loop	match	shift
KMP [1]	*none*	*fwd*	*kmp*
BM.ORIG [2]	*none*	*rev*	*d1^d2*
BM.FAST [2]	*fast*	*rev*	*d1 ^ d2*
SFC [6]	*sfc*		*inc*
SLFC [6]	*slfc*	*fwd*	*inc*
QS [13]	*none*	*fwd*	*sd1*
MS [13]	*none*	*ms*	*sd1 ^ sd2*
OM [13]	*none*		*sd1 ^ sd2*
LFOC [11]	*slfc*	*fwd+g*	*inc*

Below, we describe the implementation and performance for the components mentioned above.

4.1 Skip loops

4.1.1 *none*

This is the easiest and slowest skip loop to program, although selection of a good shift function, such as the BM $delta_1$ or the Sunday $DELTA_1$, can partially compensate for the speed loss. The simplest algorithm exemplifying this is Sunday's QS, 'quick search', which is { *none* |*fwd* | *sd* 1 }.

4.1.2 sfc

Horspool described a SFC ('Search First Character') algorithm, [6] a simple skip loop that looks for the first character of *pat*. The outer and skip loop, after adding a sentinel for speed, are

sfc

```
ch = pat [0];
text [textlen] = ch; /* install sentinel */
i = 0 ;
while (i < textlen-patlen) {
     while (text [i] != ch)    /* skip loop */
          i++;
     if (i >= textlen) break;
     match(pat [1.. ], text [ i+ 1..]);
     i += shift(i,p);
}
```

As this skip loop is often supported in hardware, we also measured *sfcm*, a version that uses the C library routine *memchr* to find *ch*. Although *memchr* is presumably as efficient as possible, all of the compilers we measured made a subroutine call to *memchr* rather than inserting the code inline.

4.1.3 slfc

Horspool also described a variant of *sfc* called SLFC ('Search Least Frequent Character'). It uses the frequency distribution of *text* to search for the least frequent character in *pat*, rather than the first character. Kowalski and Meltzer also applied this idea in their LFOC ('Least Frequently Occurring Character') algorithm. [11]

slfc

```
ch = pat [rare] ;       /* least frequent char of pat */
text[textlen] = ch; /* sentinel */
i = rare;
while (i < textlen-patlen+rare) {
    while (text[i] != ch)    /* the skip loop */
        i++;
    if (i >= textlen) break;
    match(pat [0..], text [ i-rare.. ]);
    i += shift(i,p);
}
```

The *slfcm* variant uses *memchr* to find *ch.*

4.1.4 fast

This is the first of three components based on the fast implementationof the Boyer-Moore algorithm. Boyer and Moore reported that most of the execution time of a string search is spent moving the pattern past immediate mismatches. They described a modified algorithm, which we call BM.FAST, with an initial 'fast loop' that quickly skips past these mismatches. To prevent two tests in the skip loop, one for end of text and one for a possible match, this loop uses $delta_0$, identical to Boyer-Moore's $delta_1$except that $delta_0[pat[patlen-1]]$ contains a sentinel value LARGE (> *textlen* + *patlen*) that causes the skip loop to halt when a possible match position occurs. The earliest C implementations known to us were King and Macrakis [14] in 1985 (part of GNU *emacs*) and Woods [15] in 1986 (a new version of *grep*). For performance reasons, the code is written with character pointers rather than array references; the outer and skip loops are

fast

```
s = text + patlen - 1;
e = text + textlen;
while (s < e) {
    while ((s += d0 [*s]) < e)    /* skip loop */
        ;
    if (s < e+patlen)    /* end of text */
        break;
    s -= LARGE;
    match(pat [0..patlen - 2] ,text [s -patlen + 1..s- 1 ]);
    i += shift(i,p);
}
```

Most C compilers, with optimizing enabled, generate very good code for this fast skip loop. For example, on the vax, it is four machine instructions.

4.1.5 ufast

Using the LARGE value in *fast* as a sentinel to break out of the skip loop causes problems on segmented architectures and with arithmetic overflow on pointer addition. In addition, the new C standard [16] specifies pointers may be compared only if they point inside or just after the same array of characters. These problems are both avoided by an alternate scheme devised by Haertel [17] for the Free Software Foundation's *egrep,* although his primary moti-

vation was to support loop unrolling. [18] It uses a sentinel of zero (and thus is identical with BM's *delta* $_1$); the skip loop becomes:

```
k = 0 ;
while ((k = d0[*(s += k)]) && (s < e))

if (s >= e)
     break;
```

Unfortunately, because of the double test, this runs a little slower than the *fast* loop. The speed can be recovered by a couple of standard tuning tricks such as described by Bentley. [19] We can remove one test by inserting *patlen* copies of *pat* [*patlen* – 1] at *text* [*textlen*] and so only need to test for the end of text when *k* is zero. We can gain speed by amortizing the loop overhead by unrolling the loop as it is harmless to repeat the step statement `k = d0 [ * (s += k) ]` after we hit a potential match (when *k* = 0). The benefits of unrolling are substantially dependent on the length of the patterns and the system design, for example, the size of the instruction cache. After measuring different unrolling factors (with match= *fwd,* shift= *inc),* we picked 3-fold unrolling as the best compromise across systems.

Algorithm	Execution Speed (MB/s)						Statistics	
	386	spare	mips	vax	68k	cray	step	cmp+jump
unroll1	1.81	6.83	11.91	5.30	4.02	8.37	5.22	202620 (20.3%)
unroll2	2.42	7.06	12.45	5.63	4.12	9.05	5.08	207753 (20.8%)
unroll3	2.66	7.12	12.54	5.84	4.20	9.21	4.95	213048 (21.3%)
unroll4	2.79	7.08	12.48	5.84	4.20	9.16	4.82	218468 (21.8%)
unroll5	2.84	7.00	12.39	5.87	4.15	9.04	4.69	224007 (22.4%)
unroll6	2.86	6.92	12.25	5.79	4.11	8.90	4.57	229648 (23.0%)

This finally gives us the *ufast* skip loop

ufast

```
s = text + patlen – 1;
e = text + textlen;
memset (e, pat [patlen–1] , patlen) ;
while (s < e) {
     k = d0 [*s];
     while(k)  {
          k = d 0 [ * ( s + = k ) ] ;
          k = d 0 [ * ( s + = k ) ] ;
          k = d 0 [ * ( s + = k ) ] ;
     }
     if (s >= e)
          break;
     match(pat [0..patlen –2], text [ s –parlen + 1 ..s – l]);
     s += shift(s, p);
}
```

4.1.6 lc

While investigating the performance of the algorithms described in this paper, it became clear that a large part of the performance of the fast BM algorithms depends on how long the algorithm stays in the skip loop. Since the pattern can be scanned in any order, we suddenly

realized that the skip loop character could be any character of the pattern, and not necessarily *pat* [*patlen* – 1] as in *fast.* Having a skip character near the end of the pattern gives large skips, but it might be better to choose a lower frequently character and stay in the loop longer, even though each skip is a bit smaller. Consider searching for the word 'baptize' in English text. Quite often, the 'e' will match and we will break out of the skip loop, and then do the match and shift before going back to skipping. However, if the skip loop was looking for the 'z', it would skip a little less on each iteration of the loop, but would almost never break out of the skip loop, and would thus run faster.

The algorithm that we describe here estimates the search cost associated to each character of *pat,* and selects the one with the least cost for the skip loop. Matsumoto [20] has also developed a scheme to first search for a subpattern of *pat* which minimizes the search cost, but the details of how he estimates this cost are not known to us.

We measured the possible benefits of this scheme by recording the execution times on the mips system for looping on *pat* [*patlen* – 1 – *offset*] where $0 \leq offset < min$ (12 ,*patlen*). Figure 1 shows group statistics of the fastest *offset* loop, when $offset \neq 0$, for sets of 200 words of lengths 4 through 13 (lengths 1, 2, and 3 were always fastest with *offset* = 0). The top graph shows the distribution of speed ups over *fast* (with *offset* = 0) measured for various values of *offset* and *patlen*. It has points for the minimum and maximum, a box between the 25% and 75% quartiles and a mark on the median. The bottom graph shows the fraction of all words fastest at a particular *offset,* excluding *offset* = 0 which accounts for the balance.

The potential benefit of *lc* seems small but significant. Even though the average benefit is small (about 3-5%), there are occasional speedups of almost 15%. For the case of alphabets with a small variance in the character frequencies, we would expect *lc* to be less effective. Otherwise, for the case of very long patterns, we expect *lc* to be more effective as there is more chance of using a low frequency character, and differences in skip size for different characters decrease toward the end of long patterns.

Constructing a skip loop for a particular character, say *pat [j], is* straightforward; simply calculate $delta_0$ as though *pat* was only *j* + 1 characters long. This means that the sentinel is now *d* 0 [*pat* [*j*]] = 0, and one must add *patlen* copies of *pat* [*j*] to the end of the *text* so that the search will terminate correctly.

To select the best skip character, we need to estimate the cost of running the algorithm, measured as execution time, for each *pat* [*j*], $0 \leq j <$ *patlen,* and choose the *j* that gives the 'least cost'. Let c_j denote *pat* [*j*], *p(c)* be the probability that *c* occurs in the input text, $P_j = p(c_j)$, t_1 be the time for one skip loop iteration, and t_m, be the average time to determine if a match really occurred and then shift before restarting the next skip loop. Next, for a skip loop on c_j, let sd_j be the expected skip distance (step) per mismatch. The total expected number of skip loop iterations can be estimated by es_j

$$es_j = \frac{textlen}{sd_j}$$

Also, the expected number of possible match positions is estimated as em_j

$$em_j = \frac{textlen \times P_j}{sd_j}$$

Then, the expected time (et_j) to process the whole text is given by

$$et_j = t_1 es_j + t_m em_j = \frac{textlen\ (t_1 + t_m P_j)}{sd_j}$$

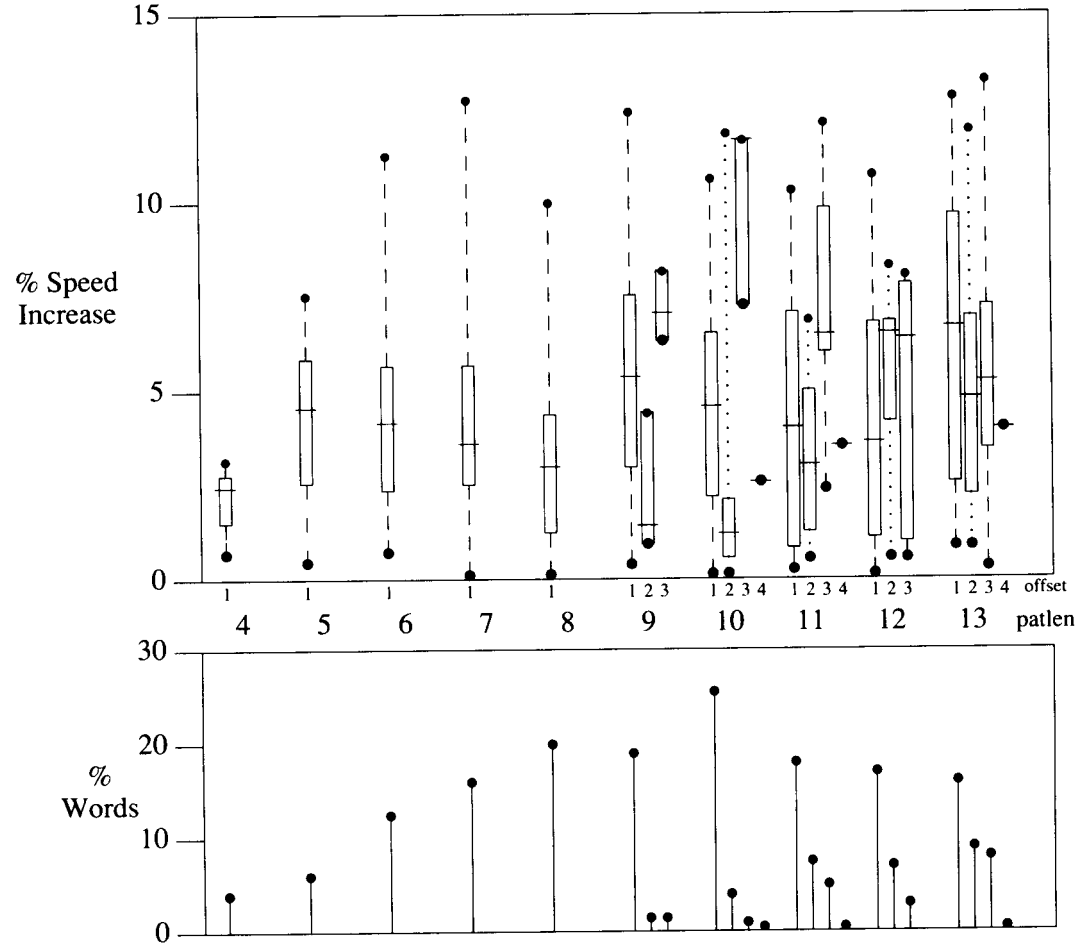


Figure 1. Potential benefits of a least cost algorithm

Let $t_{slow} = t_m/t_l$. The normalized time to search the whole text is nt_j

$$nt_j = \frac{et_j}{t_l \times textlen} = \frac{1 + P_j \times t_{slow}}{sd_j}$$

We then select the skip loop on the c_k for which nt_k is minimum.

To evaluate nt_j, we need the values of P_j, t_{slow}, and sd_j. P_j is given by the character frequency distribution for the input text. Most often, this will be approximated by a distribution empirically derived by measuring a sample of prototypical text; for example, a sample of English text.

The value of t_{slow} must be measured by running a calibration program; it depends on the choice of skip loop, match algorithm, shift function, compiler, and hardware architecture. We measured t_l and t_m by running { *ufast* | *fwd* | *inc* } with *delta* $_0$ set to all 1's and all 0's respectively. In the former case, we stay in the skip loop until the end of text. In the latter case, the skip loop always broke immediately, and was never executed. In both cases, there were *textlen* iterations executed, and t_{slow} was simply computed as the ratio of the two times measured. The measured values are

	386	spare	mips	vax	68k	cray
t_{slow} =	4.91	3.04	3.29	3.86	2.97	3.34

It is possible to derive a fomal estimate for sd_j (see Schaback [21] and Baeza-Yates [12]). On the other hand, it is easy and more accurate to derive these values directly from *text* itself. The method that gave us the most reliable estimates for the sd_j involved running a large number of searches on *text* and measuring the actual shifts that occurred. The mean and standard deviations for sd_j are

j	$m(sd_j)$	$SD(sd_j)$	j	$m(sd_j)$	$SD(sd_j)$	j	$m(sd_j)$	$SD(sd_j)$
0	1.00	0.00	5	5.29	0.19	10	8.65	0.40
1	1.96	0.03	6	6.02	0.25	11	9.25	0.43
2	2.85	0.06	7	6.73	0.29	12	9.88	0.49
3	3.73	0.09	8	7.40	0.35	13	10.55	0.54
4	4.54	0.14	9	8.03	0.35	14	11.04	0.79

These values are roughly comparable to the theoretical ones given by Schaback. The difference might be attributed to the alphabet, since we used a larger one with different character frequencies, or to higher-order statistical properties of text. In any case, estimates for the sd_j values are always going to be sloppy since the observed standard deviation is large and appears to keep increasing with j. However, the difference between successive sd_j values decreases as j increases and we can show theoretically that the mean sd_j approaches an upper bound of (| *alphabet* | – 1). Asymptotic behavior of the standard deviation is unknown to us.

Finally, we checked how well our model for estimating nt_j performed on our test set. As measured on `mips`, for 89.3% of the test set, the speed of the predicted offset was the optimal speed. For 99.0% of the test set, the speed of the predicted offset was at least as fast as *offset* = 0, that is, at least as fast as *fast* and *ufast*.

4.1.7 Summary

The performance figures for the various skip loop components were measured with *match=fwd* and shift= *inc* (so as to maximise the work done by the skip loop). Here, as in the other summaries below, different match and shift components would yield slightly different numbers.

Algorithm	Execution Speed (MB/s)						Statistics	
	386	spare	mips	vax	68k	cray	step	cmp+jump
none	0.15	1.09	2.50	0.57	0.47	0.78	1.00	1041392 (104.1%)
sfc	0.58	2.99	6.09	1.83	1.36	2.28	1.00	1041397 (104.1%)
sfcm	0.97	1.77	4.24	3.13	0.92	2.84	1.00	1041397 (104.1%)
slfc	0.62	3.18	6.27	1.92	1.43	2.39	1.00	1023043 (102.3%)
slfcm	1.29	1.90	4.54	4.08	1.07	5.23	1.00	1023044 (102.3%)
fast	2.42	6.73	10.92	5.13	3.41	7.68	5.22	202619 (20.3%)
ufast	*2.66*	*7.11*	12.52	5.81	4.21	9.21	4.95	213048 (21.3%)
lc	2.27	7.13	12.57	5.71	4.28	9.23	4.93	212909 (21.3%)

Provided the preprocessing to find the least frequent character is not onerous, *slfc* is better than *sfc*. The benefits of using the *memchr* library routine are quite system specific. For example, there are no special character search/compare instructions on the Cray. Rather, the routines are vectorized to handle the text a word at a time instead of a character at a time, and is slower unless the least frequent character occurs sufficiently infrequently for the vectorizing to help.

The BM derived skip loops *(fast, ufast,* and *lc)* are clearly superior. The *ufast* loop seems the best balance between ease of coding, portability, and optimal performance. The *lc* loop has a more complicated preprocessing routine; it runs a little faster on average and occasionally is substantially faster.

There are additional fine tuning refinements that may be architecture dependent. One such variant changes the type of the *d0* skip table from `int` to `char`. As shown below, this runs faster on the two RISC architectures (`mips, sparc`) but slower on the others. Examination of the generated code reveals that the `mips` code had shrunk by one instruction because it no longer had to multiply the index by 4 to get a byte address for a skip table entry. The `vax` code, on the other hand, grew because it cannot add a byte to an integer directly. The following execution times were for match= *fwd* and shift= *inc.*

Algorithm	Execution Speed (MB/s)						Statistics	
	386	sparc	mips	vax	68k	cray	step	cmp+jump
ufast	2.66	7.11	12.52	5.81	4.21	9.21	4.95	213048 (21.3%)
ufast+c	2.40	7.68	13.57	5.07	3.85	7.46	4.95	213048 (21.3%)
lc	2.27	7.13	12.57	5.71	4.28	9.23	4.93	212909 (21.3%)
lc+c	2.14	7.71	13.62	4.54	3.89	7.47	4.93	212909 (21.3%)

4.2 **Match Algorithms**

In this section, we describe different techniques for determining if *pat* fully matches *text* at its current position. After all but the *none* skip loop we know that a particular text character already matches; most match algorithms can use this information. The choice of match algorithm is often dependent on the shift function one wants to use. Some shifts, for example *d1* and *d2,* require knowing the rightmost mismatch point, and so one must use the *rev* match.

4.2.1 fwd

The most straightforward match algorithm simply compares *pat* [*k*] with *text* [*i+ k*] for *k = 0 , l ,... ,patlen* – 1, yielding the leftmost mismatch. If any characters, such as the skip or *guard (see* below) character, are known to match at the beginning (or end) of *pat,* then the bounds are adjusted accordingly. However, if a known matching character is in the middle of *pat,* it may be faster to compare the whole of *pat* instead of comparing the disjoint parts before and after the matching character(s). This is particularly true when there is special system support for *fwd,* either as library routines such as C's *memcmp* or even hardware instructions for character comparison. We also timed *fwdm,* the version of *fwd* that uses *memcmp.*

4.2.2 rev

The reverse match algorithm *rev* is the obvious counterpart *fwd,* scanning *pat* backward (*k = patlen* – 1 ,..., 1,0), and yielding the rightmost mismatch. If any characters, such as the skip or guard character, are known to match at the beginning (or end) of *pat,* then the bounds are adjusted accordingly. It is rare for a system to have special support for reverse character comparison.

4.2.3 om

As observed by Kowalski and Meltzer, [11] Baeza-Yates, [12] and Sunday, [13] given the alphabet frequency distribution, we will minimize the expected number of match comparisons if they are done in ascending frequency order. However, the implementation normally involves indirection and runs slower on some architectures. Sunday's implementation used the following structure

```
struct {
    int loc;
    char c;      /* c == pat[loc] */
} om[MAXLEN] ;
```

where *om* [ ] represents the pattern sorted in increasing frequency order. Let *s* point at the text position *text* [*i*] corresponding to *pat* [0]. Then the *om* match can be coded as

om

```
    for (p = om; p < om+patlen; p++) {
        if (p->c != s [p->loc] )
            goto  mismatch;
    }
    /* match found */
mismatch:
    i += shift (i, p->loc) ;
```

4.2.4 guard

For our test set, the average number of characters compared in the above *om* loop is 1.05. Thus, we can gain 95% of the benefits of the *om* match by simply testing for the rarest character of the pattern first before doing a full match test. This 'guard test' is a simple piece of code. The preprocessing step determines *rarest* such that *par* [*rarest*] is the *par* [*k*] (other than the skip loop character) with lowest frequency value. Then, the guard test is

guard

```
    if (text [i+rarest] ! = pat [rarest] )
        goto  mismatch;
```

match

Except for being different than the skip loop character, the guard is independent of the skip loop and match algorithm and can be combined with any other match algorithm, although it is redundant with *om.* A guard is denoted by '+ *g*', so *fwd* + *g* is a guard test before a *fwd* match scan. Use of the guard generally abrogates properties of the match algorithm such as knowing the rightmost mismatch.

Subsequent to our discovery of the guard, we found an earlier reference to the use of a guard in Kowalski and Meltzer's LFOC algorithm. [11] They also discuss the notion of multiple guards which extends naturally into the *om* algorithm.

4.2.5 ms

The maximal shift algorithm [13] *ms* uses a scan order that gives the largest possible *sd2* shifts. It seems only effective for long patterns and thus we do not consider it further, except to note that the match function would be coded similarly to the *om* match test. One can also add a guard for a *ms+g* match test.

4.2.6 *Summary*

The match algorithms were measured with skip= *ufast* and shift= *inc* (the preferred skip loop and simplest shift).

Algorithm	Execution Speed (MBIs)						Statistics	
	386	sparc	mips	vax	68k	cray	step	cmp+jump
fwd	2.66	7.11	12.52	5.81	4.21	9.21	4.95	213048 (21.3%)
fwdm	2.59	6.56	12.14	4.78	3.23	8.31	4.95	213048 (21.3%)
rev	2.66	6.97	12.60	5.80	4.14	9.09	4.95	215464 (21.5%)
om	2.32	7.06	12.37	5.78	4.15	9.48	4.95	212791 (21.3%)
fwd+g	3.04	7.30	12.64	6.10	4.35	9.67	4.95	203226 (20.3%)
fwdm+g	3.04	7.25	12.85	5.87	4.12	9.57	4.95	203226 (20.3%)
rev+g	3.04	7.30	12.65	6.12	4.34	9.66	4.95	203425 (20.3%)

The *fwdm* match using *memcmp is* uniformly inferior, which is not surprising as the average number of characters compared was measured to be 1.08. *Fwd* and *om* have similar performance but *fwd is* preferred as it is easier to implement. Moreover, combining *fwd* with a guard is a clear winner.

4.3 **Shift functions**

In this section, we describe different techniques for determining how far to shift position in the *text* after we finish the match loop. Important characteristics of a shift function are worst case performance, the cost of preprocessing, and assuming that *match* establishes *text* [*i* +*p*] *#pat* [*p*] when it fails, what properties of *i* and *p* are needed. Only *kmp, d2* (and combinations of *d2* such as *d12),* and *gd2* guarantee linear worst case performance. *Sd2* is conjectured to have linear worst case performance. The other functions have a worst case performance of *O* (*textlen* × *patlen).*

4.3.1 inc

This simply increments the current text pointer by one. It uses neither i nor *p,* and can be used with any skip loop and match algorithm. With skip= *none* and match= *fwd,* using *inc* yields the obvious (and slow) straightforward algorithm used by many programmers. However, with a non-null skip loop, the bulk of the text is scanned by the loop and the shift function is of less importance. In this case, *inc is* a good choice because it executes so quickly.

4.3.2 kmp

The Knuth-Morris-Pratt [1] shift function *kmp* requires knowing the leftmost mismatch. If *text* [*i* + *p*] is the leftmost mismatch, then the characters *text [i.. i* +*p* – 1] match *pat* [*0* ..*p* – 1]. We can thus shift *pat* right by the minimum amount *k* such that *pat* [*0..p* – 1 – *k*] is matched with the prior *pat* [*k..p* – 1] which was aligned with *text [i*+ *k.. i* +*p* – 1], and a new character is brought into position with *text* [*i* +*p*]. These shifts *kmp* [*p*] can be computed from *pat* as a preprocessing step before the search begins.

If one wants to preserve the linear behavior of *kmp,* one needs to use a variant of *fwd* that avoids backtracking in *text* by starting at the previous mismatch location. This *fwd* variant with the *kmp* shift is the original KMP algorithm, which is known to be slower than classic BM ,[7] and so we do not consider *kmp* in this paper.

4.3.3 dl

This is the BM *delta* $_1$ and requires knowing the rightmost mismatch in *pat.* The initialization of *delta* $_1$ is

```
for(c = 0; c < alphabetize; c++)
    d1 [c] = patlen;
for(i = O; i < patlen; i++)
    d1[pat [i]] = patlen–1–i;
```

If the rightmost mismatch is between *pat* [*p*] and *text* [*i* +*p*], then the actual BM *delta* $_1$ pattern shift is *delta* $_1$[*text* [*i* +*p*]] – (*parlen* – 1 –*p*). When this is < 0, 1 must be used. The shift is

```
i += max (d1 [text [i+p]] , patlen-p) ;
```

4.3.4 d2

This is the BM *delta* $_2$ and requires knowing the rightmost mismatch in *pat.* It is an adaptation of the *kmp* shift from the *fwd* to the *rev* match scan. Using *d2* also gives a linear bound on the worst case search behavior, although this bound is worse than *kmp'* s since it cannot avoid retesting already matched input. However, it will usually give larger shifts than *kmp.* Initializing the *d2* table in *O* (*patlen*) time is nontrivial; the algorithm in KMP [1] gives incorrect shifts, the algorithm in Smit[7] does not give optimal shifts and the algorithm in Sunday [13] runs in *O* (*patlen* ^{2}) time. We based ours on Rytter. [22]

4.3.5 d12

This is the shift function used by Boyer and Moore. It is simply the maximum of *delta,* and *delta* $_2$ as defined above:

```
i += max(d1 [text [i+p] ] , d2 [p] ) ;
```

Because the *delta* $_2$ shift is always > 0, we need not worry about the *d1* shift being < 0.

4.3.6 sd1

Sunday [13] observed that in algorithms based on BM, given that the right hand end of *pat* is aligned with *text* [*i* +*patlen* – *1*], then some character in the next position of *pat* must align with *text* [*i* +*patlen*], and thus we can define a shift function *DELTA* $_1$ based on *text* [*i* +*patlen*]. *DELTA* $_1$ *is* computed as (*delta* $_1$ + 1) but its application is independent of *p,* and unlike *d1,* the shift is always > 0, and can be used directly:

```
i += sd1 [text [i+patlen]] ;
```

This shift is noteworthy because it does not depend on any information from the match test and thus, arbitrary and nonsequential scan orders can be used with it.

4.3.7 sd2

This is Sunday's generalization [13] of the BM *delta* $_2$ for a match algorithm using an arbitrary scan order of the pattern. Sunday conjectures that *sd2* ensures linear worst case search behavior. For match= *fwd* (or *rev),* it is the same as shift= *kmp* (or *d2),* and so we do not consider it here. For the patterns considered in this paper, a condensed version of *sd2* (*md2*) *is* sufficient.

4.3.8 md2

If we use a skip loop (other than *none),* then *text* [*i* + *k*] =*pat* [*k*] where *pat* [*k*] is the skip loop character. A mini *sd2,* or *md2,* shift aligns the rightmost occurrence of *pat [k]* left of *pat* [*k*] with *text* [*i* + *k*]. If there isn't one, the shift is *k* + 1. This shift is simple to precompute and should always outperform *inc* since it is as easy to apply but typically will be much greater than 1 and roughly equal to sd_k, the distance that the skip character must skip to match itself. On average (for our test set), *patlen* ≈ 7, *k*≈ 6 and *md2* = 6.23, which is close to sd_6=6.06. A corresponding shift can also be defined when multiple characters are known to match, such as the skip and guard characters, but we have not investigated this case.

4.3.9 gd2

The *gd2* shift is a generalization of the $delta_2$ shift by Giancarlo.[23] It incorporates the *delta,* character heuristic into BM's $delta_2$ heuristic. The BM $delta_2$ brings a new pattern character different from *pat* [*p*] into correspondence with the text char *text* [*i* +*p*] at which the mismatch occurred, since if it was the same as the previous one it would just mismatch again. Giancarlo's shift extends this by having the new character be exactly the same as *text* [*i* +*p*] since we already know it. To do this, one needs to have an array of shifts indexed by the mismatch position *p,* and the text alphabet character *a* at that position. Let *m = patlen;* then

gd2 [*p*][*a*] = *m*– 1 –*p* + *min* { k|(k>0)

and ((*k*>*i*) **or** (*pat* [*i*– *k*..*m*– 1 –*k*]= *pat* [*i*..*m*- 1], for *p*<*i*<*m*))

and ((*k*>*p)* **or** ((*pat* [*p* –*k*] =*a*)#pat[p])) }

This function is computed in *O (m | alphabet |) time and space; the preprocessing time is* substantially greater than the other shift functions. It is easy to show *gd2* [*m*– *1*][a]= *delta* 1 [*a*], and *gd2* [*p*][*a*] *2max* (*delta* 1[*a*], *delta2* [*p*]) for *O*<*p*<*m*– 1. The *gd2* shift function performs well on our test set, but as we shall see below, is astonishingly fast for long patterns.

4.3.10 multiple

If one shift is good, then perhaps combining two or more shifts, for instance by using their maximum, would be better. In general, it seems not; the overhead involved in doing this appears to destroy any advantage one gains in isolated instances. This can be seen in timing comparisons of the *d* 1 ^ *d2* (= *d* 12) or *sd* 1 ^ *md2* shifts and is true whether the maximum is calculated at execution time with *if* statements or by a precomputed two-dimensional array.

4.3.11 Summary

A small example illustrates the difference between some of these shifts. Suppose we are searching for the word `contention` where the skip character is the final `n`. Using the rev match, we find a mismatch at `i`. For the text below, the resulting different shifts are

```
        contention
usurped  the  throne
         contention          inc = 1
                contention   dl(r)= 8
               contention    d2(7)= 7
              contention     sd1(e)= 6
             contention      md2= 4
                    contention  gd2(7,r)= 10
```

One interesting shift we did not evaluate is that described by Baeza-Yates. [12] It is similar to the l_C skip loop but applied to the shift function rather than to the skip loop and minimizes $j\,(1 - P_j)$ rather than nt_j.

The following timings for the shift component have skip= *ufast* and match=rev(thepreferred skip loop and some of the shifts need a rightmost mismatch).

Algorithm	Execution Speed (MB/s)						Statistics	
	386	sparC	mips	vax	68k	cray	step	cmp+jump
inc	2.66	6.97	12.60	5.80	4.14	9.09	4.95	215464 (21 .5%)
d1	2.01	6.82	12.46	5.56	4.02	8.84	5.12	218923 (21 .9%)
d2	2.05	6.84	12.36	5.57	3.99	8.82	5.00	223531 (22.4%)
d1^d2	2.44	6.80	12.32	5.49	3.88	8.88	5.14	218203 (21.8%)
sd1	2.68	7.02	12.72	5.87	4.18	9.11	5.22	204908 (20.5%)
md2	2.72	7.19	12.96	5.97	4.31	9.46	5.18	216368 (21.6%)
sd1^md2	2.70	7.06	12.78	5.91	4.21	9.21	5.25	203401 (20.3%)
gd2	2.44	6.93	12.47	5.44	3.89	8.85		

the shifts and systems we measured, *md2* is the fastest shift. It is fast and compact, easy to precompute as a constant, and minimizes overhead in the search loop.

5. RECOMMENDED SEARCHING ALGORITHMS

For general purpose use, we recommend the two algorithms TBM and LC described below. Their components were selected as consistent front runners across the board on all the architectures we evaluated. Their C implementations are reproduced in Figures 2 and 3 and are structured as a preprocessing and an execution routine. The text frequency distribution and values for the sd_i are not shown. The C source is also available electronically (see the next section for details).

The TBM (Tuned Boyer-Moore) algorithm is { *ufast* | *fwd* + *g* | *md2* }. It is compact and fast and can be made independent of frequency data by eliminating the guard.

The LC (Least Cost) algorithm is { l_C | *fwd* + *g* | *md2* }. It differs from TBM only in the skip loop used; the l_C loop makes it a bit faster in some instances. When no text frequency data is available, this reduces to `TBM`.

Algorithm	Execution Speed (MB/s)						Statistics	
	386	sparc	mips	vax	68k	cray	step	cmp+jump
TBM	3.11	7.51	12.98	6.25	4.53	10.08	5.18	204199 (20.4%)
TBM-g	2.72	7.32	12.89	5.98	4.39	9.58	5.18	214008 (21.4%)
LC	2.45	7.54	13.10	6.31	4.57	10.06	5.16	204592 (20.5%)

When comparing these numbers to the previous ones for *ufast* and *lc*, the reader will note some anomalies. Unfortunately, the statistics for any algorithm cannot be deduced from the sum of the components, since they are perturbed in combination. For example, the LC skip

```
#include     " f req. h"

#define ASIZE 256

static struct
{
    int patlen;
    unsigned char pat[1024];
    long  delta[ASIZE];
    int rarec, rareoff, md2;
} pat;

void prep(unsigned char *pb, int len)
{
    register unsigned char *pe, *p;
    register int j, r;
    register long *d;

    pat.patlen = len;
    assert(pat.patlen < sizeof(pat.pat));    /* true or abort */
    memcpy(pat.pat, pb, pat.patlen) ;     /* store pattern */
    /* get skip delta */
    for(j = 0, d = pat.delta; j < ASIZE; j++)
        d[j] = pat.patlen;
    for(pe = pb+pat.patlen–1; pb <= pe; pb++)
        d[*pb] = pe–pb;
    /* get guard */
    r = 0 ;
    for(pb = pat.pat, pe = pb+pat.patlen–1; pb < pe; pb++)
        if(freq[*pb] < freq[pat.pat[r]])
            r = p b  – pat.pat;
    pat.rarec = pat.pat[r];
    pat.rareoff = r – (pat.patlen–1);
    /* get md2 shift */
    for (pe = pat.pat+pat .patlen-1, p = pe–1; p >= pat.pat; p––)
        if(*p == *pe) break;
    /* *p is first leftward reoccurrence of *pe */
    pat.md2 = pe – p;
}
int exec(unsigned char *base, int n)
{
    register unsigned char *p, *q, *e, *s, *ep;
    register long *d0 = pat.delta;
    register k, n1 = pat.patlen–1, nmatch = 0, md2 = pat.md2;
    register ro = pat.rareoff, rc = pat.rarec;

    s  =  base+pat.patlen–1;
    e = base+n;
    ep = pat.pat + pat.patlen–1;
    memset(e, pat.pat [pat.patlen–1] , pat.patlen); /* sentinel */
```

Figure 2A. The code for TBM

```
    while(s < e) {
        k = d0 [*s];                              /* ufast skip loop */
        while(k) {
            k=d0[*(s+=  k)];
            k=d0[*(s+=  k)];
            k=d0[*(s+=  k)];
        }
        if(s >= e)
            break;
        if(s[ro] != rc)                           /* guard test */
            goto mismatch;
        for(p = pat.pat, q = s-n1; p < ep; ){     /* fwd match */
            if(*q++ != *p++)
                goto mismatch;
        }
        nmatch++;
    mismatch:
        s += md2;                                 /* md2 shift */
    }
    return(nmatch) ;
}
```

Figure2B. The code for TBM

loop is more likely to loop on the least frequent character of par than in TBM, and thus choose a less effective guard character.

Although reconsider TBMand LC the best general purpose algorithms, they may not be optimal for a particular architecture. To illustrate this, we show below the fastest programs we found for each of the systems we tested. Note that we did not test every possible combination of components, just the promising ones as indicated by the component summary tables.

Algorithm	Execution Speed (MB/s)						Statistics	
	386	sparc	mips	vax	68k	cray	step	cmp+jump
TBM	**3.11**	7.51	12.98	6.25	4.53	**10.08**	5.18	204199 (20.4%)
LC+c	2.33	**8.15**	**14.20**	5.47	4.17	8.07	5.16	204592 (20.5%)
LC	2.45	7.54	13.10	**6.31**	**4.57**	10.06	5.16	204592 (20.5%)

If you want the fastest program for your system, you need to go through the same process of measuring the individual components and then measuring various combinations of the best components. To facilitate this, we have made a toolkit of the components described above available electronically. The authors would like to know about the best programs for various systems and about any new components, preferably by sending details by electronic mail to either andrew@research.att.com or dan@aplexus.jhuapl.edu.

6. GETTING THE CODE

All the programs, word lists and most of the testing apparatus used in this paper are available electronically via *ftp* (login as user netlib on research.att. com) or *nelib.*[24] *Netlib* works by sending electronic mail messages to a server. For example, the message

```
send index
```

```
#include     “freq.h”
#include     “sd.h”

#define ASIZE 256

static struct
{
    int patlen;
    unsigned char pat[1024];
    int  loopoffset;      /* loop on pat[ patlen-1-loopoffset] */
    long  delta[ASIZE];
    int rarec, rareoff, md2;
} pat;

void prep(unsigned char *pb, int len)
{
    register unsigned char *pe, *p;
    register int j, r, rr;
    register long *d;
    double tmax, f;
    extern double tslow;

    pat.patlen = len;
    assert(pat.patlen < sizeof(pat.pat));    /* true or abort */
    memcpy(pat.pat, pb, pat.patlen) ;
    /* get least cost skip loop char */
    tmax = 2+tslow;
    for(pb = pat.pat, pe = pb+pat.patlen; pb < pe; pb++){
        f = (1 + tslow*freq[*pb]) / sd[pb–pat.pat];
        if(tmax > f){
            tmax = f;
            r = pb–pat.pat;
        }
    }
    pat.loopoffset =  pat.patlen-1–r;
    /* get skip delta */
    d = pat.delta;
    for(j = 0; j < ASIZE; j++)
        d[j]  = pat.patlen–pat.loopoffset;
    pe = pat.pat+pat.patlen–1-pat.loopoffset;
    for(pb = pat.pat; pb <= pe; pb++)
        d[*pb]  = pe-pb;
    /* get guard  != skip loop char */
    rr = r;
    r = 0;
    for(pb = pat.pat, pe = pb+pat.patlen-1; pb <= pe; pb++){
        if(((pb–pat.pat) != rr) && (freq[*pb] < freq[pat.pat[r]])
            r = pb - pat.pat;
    }
```

Figure 3A. The code for LC

```
    pat. rarec = pat. pat[r];
    pat. rareoff = r - (pat .patlen-1-pat . loopoffset) ;
    /* get md2 */
    pe = pat.pat+pat.patlen-1-pat .loopoffset;
    for(p = pe-1; p >= pat.pat; p--)
        if (*p == *pe) break;
    /* *p is first leftward reoccurrence of *pe */
    pat.md2 = pe - p;
}

int exec(unsigned char *base, int n)
{
    register unsigned char *p, *q, *e, *s, *ep;
    register long *d0 = pat.delta;
    register k, t, nmatch = 0, md2 = pat.md2;
    register ro = pat.rareoff, rc = pat.rarec;

    t = pat.patlen-1-pat .loopoffset;
    s = base+t;          /* skip char position */
    e = base+n;
    ep = &pat.pat [pat.patlen] ;   /* end of pattern */
    memset(e, pat.pat[t], pat.patlen);          /* sentinel */
    while(s < e){
        k = d0[*s];                             /* ufast skip loop */
        while(k) {
            k=d0[*(s+=  k)];
            k=d0[*(s+=  k)];
            k=d0[*(s+=  k)];
        }
        if(s >= e)
            return(nmatch) ;
        if(s[ro] != rc)                         /* guard test */
            goto mismatch;
        for(p = pat.pat, q = s-t; p < ep; ){    /* fwd match */
            if(*q++ != *p++)
                goto mismatch;
        }
        nmatch++;
    mismatch:
        s += md2;                               /* md2 shift */
    }
    return(nmatch) ;
}
```

Figure 3B. The code for LC

will cause generic information on other packages and servers to be sent to you by return mail.

The material in this paper is available from the `stringsearch` package. The components at the time of writing are

`index`	summary of what's available
`bmsrc`	test harness and all algorithms
`bmdata`	Bible text and word list
`bmct1`	scripts to generate performance data
`bmbio`	DNA text and word lists

Both `bmdata` and `bmbio` are large; `bmsrc, bmct1,` and `bmdata` are sufficient to replicate the main performance measurements. For example, with *netlib* you would type something like

```
mail netlib@research.att.com
send bmsrc bmdata bmct1 from stringsearch
```

We have made the code and data sets available for two reasons. Firstly, having the source for these algorithms conveniently available can only raise the standard of software that does string searching. Some of the components, such as $delta_2$, are subtle and hard to implement from scratch. (In fact, of the four versions of $delta_2$ described in section 4.3.4, only Rytter is correct and optimal.) Secondly, we would like to start a trend of publishing (electronically) sufficient information to allow someone other than the author(s) of a string search paper to reproduce the results, or compare a new algorithm with an existing one on the same test data.

7. DISCUSSION

The main reason theorists use a comparison count as the primary metric for comparing algorithms is its independence of implementation or system details. For exactly the same reason, if run speed is the primary metric, you must consider details of implementation and evaluate 'obviously' inferior algorithms. For example, consider the straightforward SFC, SFCM (doing SFC'S character search with *memchr),* `and` BMORIG (the classic Boyer-Moore algorithm).

Algorithm	Execution Speed (MB/s)						Statistics	
	`386`	`sparc`	`mips`	`vax`	`68k`	`cray`	step	cmp+jump
SFC	0.58	2.99	6.09	1.83	1.36	2.28	1.00	1041397 (104.1%)
SFCM	0.97	1.77	4.24	3.13	0.92	2.84	1 .00	1041397 (104.1%)
BM.ORIG	0.36	2.38	5.04	1.10	0.95	2.05	5.42	382579 (38.3%)

For all of the systems tested, SFC is faster than BM.ORIG despite doing three times as many character comparisons. And, on `386` and `vax,` SFCM is nearly three times faster. Clearly, the run time metric is unrelated to the character comparison metric.

We will compare the following algorithms in detail. They represent the better of the previously published algorithms together with our two new algorithms.

Algorithm	skip loop	match	shift
BM.ORIG	*none*	*rev*	*d1^d2*
QS	*none*	*fwd*	*sd1*
BM.FAST	*fast*	*rev*	*d1^d2*
TBM	*ufast*	*fwd+g*	*md2*
LC	*lc*	*fwd+g*	*md2*

Running our standard test on these algorithms gives

Algorithm	Execution Speed (MB/s)						Statistics	
	386	sparc	mips	vax	68k	cray	step	cmp+jump
BM.ORIG	0.36	2.38	5.04	1.10	0.95	2.05	5.42	382579 (38.3%)
QS	0.79	4.16	8.51	2.62	2.03	3.59	6.24	172365 (17.2%)
BM.FAST	2.18	6.47	10.75	4.60	3.22	7.52	5.42	208169 (20.8%)
TBM	3.11	7.51	12.98	6.25	4.53	10.08	5.18	204199 (20.4%)
LC	2.45	7.54	13.10	6.31	4.57	10.06	5.16	204592 (20.5%)

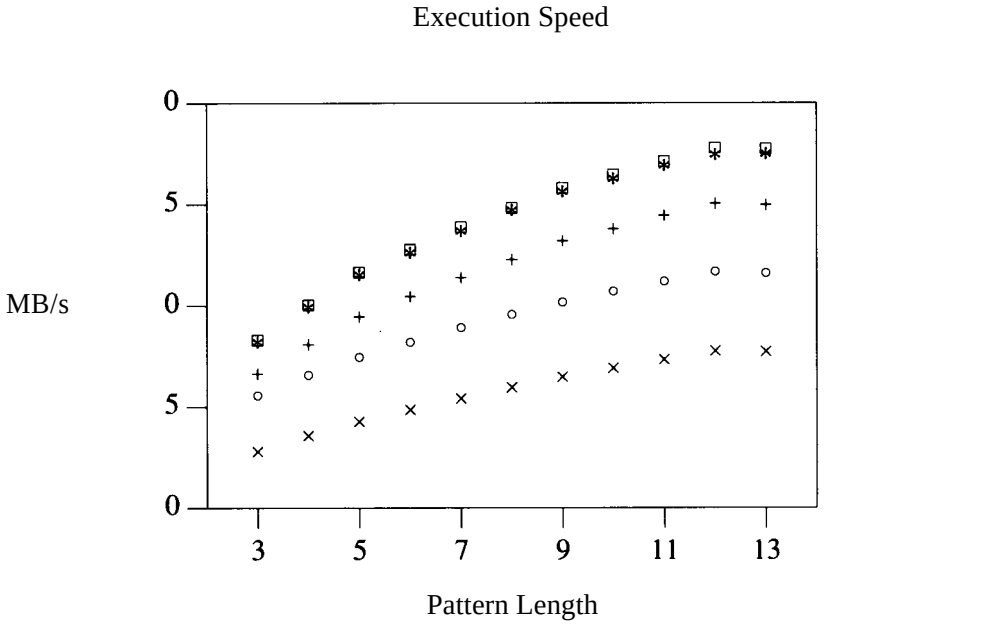
Would this comparison change if the pattern length distribution is changed? Figure 4 shows the effect of the pattern length on each of the algorithms. Each test looked for 200 randomly selected words of the specified length. The test was run on just one system (`mips`). Figure 5 reflects the same data but the performance is shown relative to BM.FAST. Performance clearly improves as the pattern length increases, and the relative merits of the different algorithms seem fairly constant within the normal usage *patlen* range. It is interesting to note that the extremely simple QS algorithm runs slower than the algorithms with skip loops despite making fewer references to text.

Various algorithms described above depend on the character frequency distribution for *text*. For the experiments run above, the distribution used was perfect — measured directly from *text;* but, in most applications, it will most likely be an estimate. For example, most of the texts searched on our UNIX systems are either documents or program sources. One could therefore take a representative sample and measure the frequencies from that sample. It is therefore important to measure the sensitivity of these algorithm's performance to the accuracy of the character frequency distributions. We examined the results for TBM and LC on the `mips` system for four different frequency distributions: (a) the distribution found in *text*, (b) the inverse order of *text*, or l–(a), (c) distribution for 7.7MB of formatted manual entries, and (d) distribution from 11.2MB of C program text. We would expect (a) to perform best, (b) to perform worst and (c) and (d) to fall in between. In fact, we found the performance was almost independent of the distribution used. The following table shows the maximum spread between the four distributions:

	speed	cmp+jump
TBM	1 .3%	1.3%
LC	1 .3%	1.1%

This is partially explained by our standard test which averages out the occasional real winners (and less frequent losers). In any case, a pragmatic approach might be to have a fixed frequency distribution and modify it on basis of a sample of *text.* This may not be worthwhile if *textlen* is small.

In this paper we have paid scant attention to long pattern strings. For long strings, say greater than 30 characters, the influence of the various $delta_2$ shifts (including *kmp, d2, sd2,* and *gd2*) dominates the other components. We can demonstrate this effect by a brief excursion into a problem domain where long patterns are commonplace. We tested some of our shift functions by searching for DNA patterns from 10 up to 250 characters long in a portion of the GenBank DNA database; 25 the results are shown in Figure 6. These results differ from the results above because of the length of the patterns, the structure (such as repeated substrings) of the patterns, and the small alphabet (5 characters: `actg` and newline). We complete our glance at long patterns by noting the tremendous speed of the *gd2* shift; it obviously merits further study.



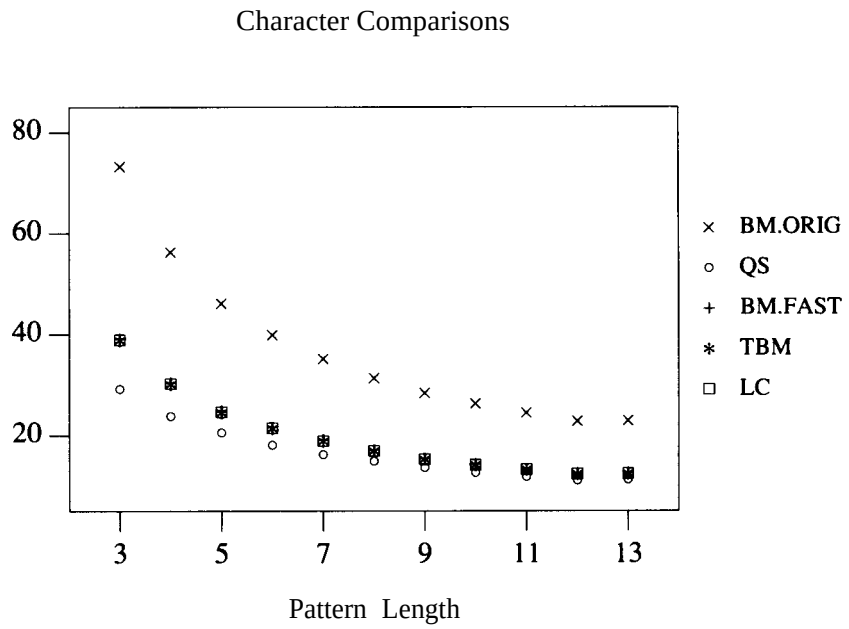


Figure 4. Absolute performance of the main algorithms

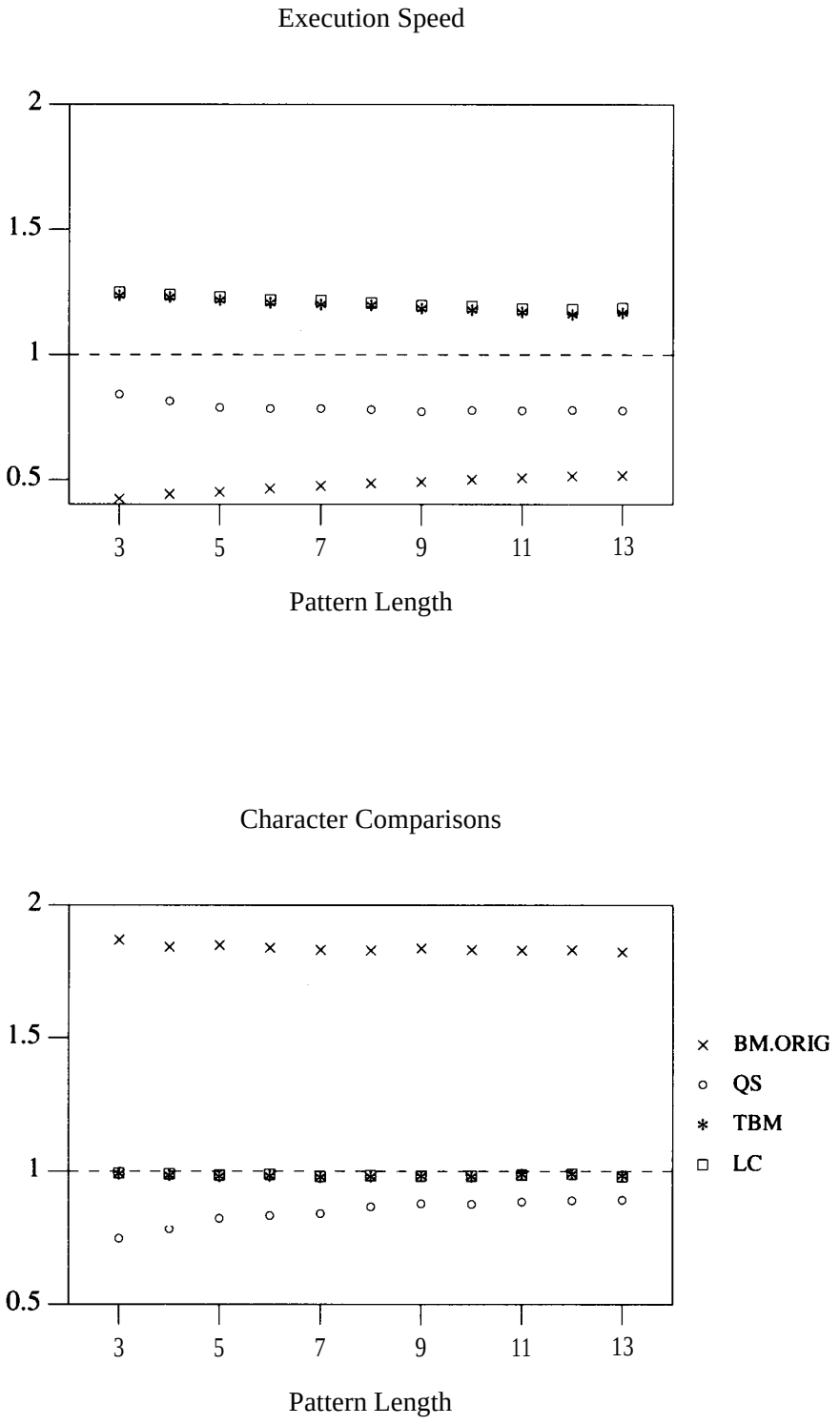


Figure 5. Performance of the main algorithms relative to BM.FAST

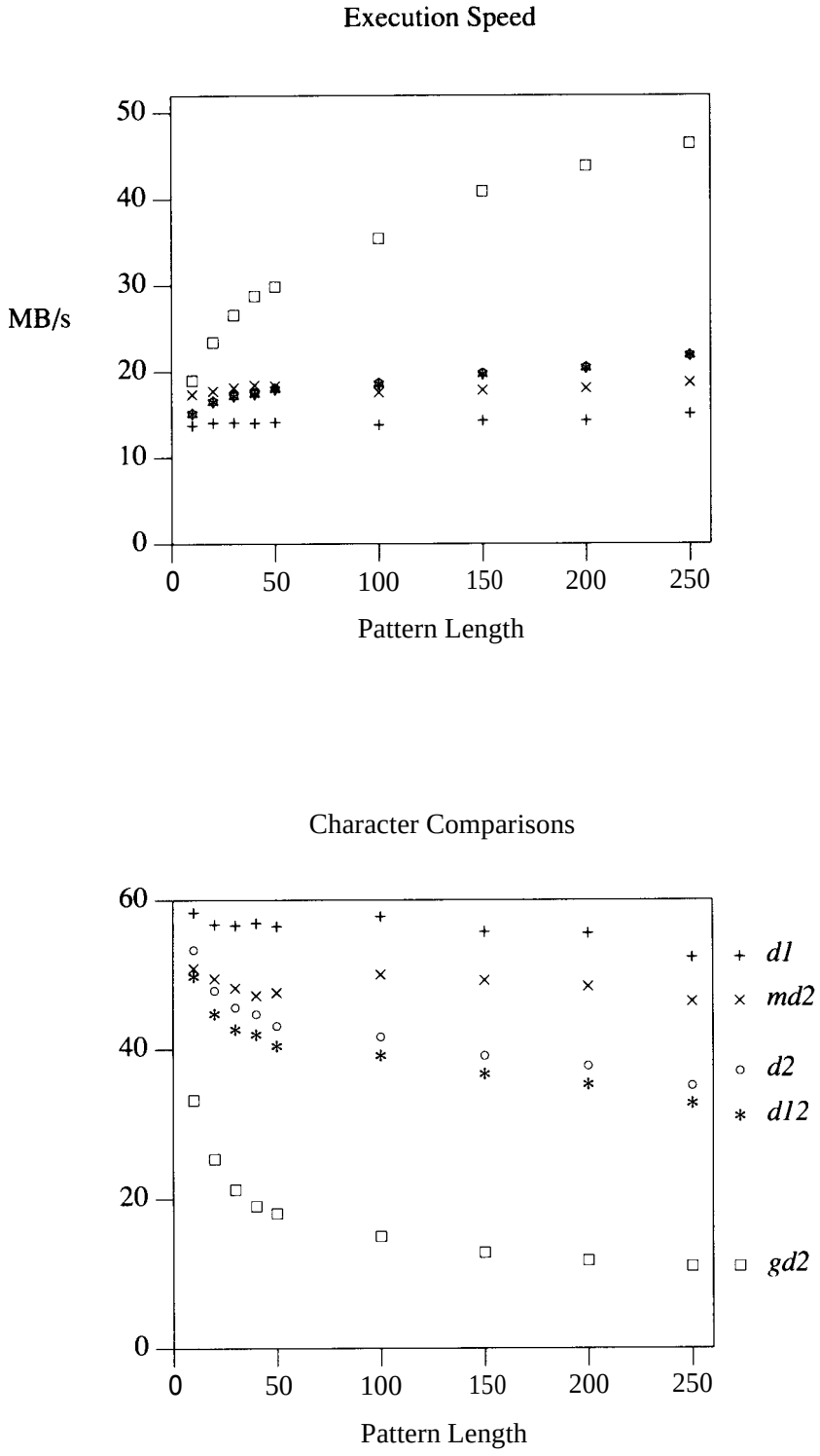


Figure 6. Performance of shift functions in searching DNA strings

8. CONCLUSION

Much practical and theoretic work on string searching has used the Boyer-Moore algorithm as the standard for comparison. Unfortunately, this work has largely ignored Boyer-Moore's alternative 'fast loop' form of their algorithm despite its better performance. We have described TBM, a portable tuned descendant of the fast BM algorithm, and LC, a variant that uses frequency distribution information. They perform about 47% fewer character comparisons and are from 2.6 to 8.6 (geometric mean of 4.5) times faster than classic BM and from 12% to 43% (mean of 29%) faster than fast BM. These algorithms are so fast that for many applications, such as the UNIX system tools *grep* and *egrep*, underlying system issues such as I/O management are now the performance bottlenecks.

Despite the superiority of these two algorithms to Boyer-Moore, they are nevertheless compromises designed for normal conditions. Different conditions or user needs may require algorithms embodying different compromises. Rather than trying to anticipate every possible set of needs, we have provided a framework for constructing algorithms based on fast BM, consisting of a *skip* loop, a *match* algorithm, and a *shift* function, and a toolkit containing several choices for each of these parts. This allows a user to systematically evaluate the various components and build an algorithm best suited to the user's application.

ACKNOWLEDGEMENTS

We'd like to thank the referees, Jon Bentley, Brian Kernighan and Stavros Macrakis for their constructive comments. James Woods sparked Hume's interest in the Boyer-Moore algorithms in 1986 and Hume's management, particularly Al Aho, has indulged his fixation ever since.

REFERENCES

1. D. E. Knuth, J. H. Morris Jr, and V. R. Pratt, 'Fast pattern matching in strings,' *SIAM J. Comput.* **6,** (2), 323–350 (1977).
2. R. S. Boyer and J. S. Moore, 'A fast string searching algorithm,' *Carom. ACM* **20,** (10), 262–272(1977).
3. R. Cole, 'Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm,' Technical Report 512, Computer Science Dept, New York University (June 1990).
4. A. Apostolic and R. Giancarlo, 'The Boyer-Moore-Galil string searching strategies revisited,' *SIAM J. Comput.* **15,** (l), 98–105 (1986).
5. L. Colussi, Z. Galil, and R. Giancarlo, 'The exact complexity of string matching,' *31st Symposium an Foundations of Computer Science* **I,** 135–143, IEEE (October 22-24 1990).
6. R. N. Horspool, 'Practical fast searching in strings,' *Software—Practice and Experience* **10,** (3), 501–506 (1980).
7. G. D. V. Smit, 'A comparison of three string matching algorithms,' *Software—Practice and Experience* **12,** (1), 57–66 (1982).
8. J. A. Woods, *More pep for Boyer-Moore grep,* Usenet netnews group `net.unix` (March 18 1986). Also Usenet archive `comp. sources. unix,` Volume 4, *egrep* (March 1987).
9. A. Hume, 'A tale of two greps,' *Software—Practice and Experience***18,** (11), 1063–1072 (1988).
10. B. W. Kernighan and D. M. Ritchie, The C *Programming Language,* Prentice Hall, Englewood Cliffs, NJ, 1988.
11. G. Kowalski and A. Meltzer, 'New multi-term high speed text search algorithms,' *First International Conference on Computer Applications,* 514–522, IEEE (June 20-22 1984).

12. R. A. Baeza-Yates, 'Improved string searching,' *Software-Practice and Experience* **19,** (3), 257–271 (1989).
13. D. M. Sunday, 'A very fast substring search algorithm,' *Comm. ACM* **33,** (8), 132–142 (1990).
14. *S.* Macrakis (February 1991). Personal communication.
15. J. A. Woods, *egrep,* Usenet archive `comp.sources.unix,` Volume 4 (March 1987).
16. ANSI, *Programming Language – C (X.?. 159-1989),* American National Standards Institute, New York, 1989.
17. M. Haertel, *GNU e?grep,* Usenet archive `comp.sources.unix,` Volume 17 (February 1989).
18. M. Haertel (December 1990). Personal communication.
19. J. L. Bentley, *Writing Efficient Programs,* Prentice-Hall, Englewood Cliffs, N. J., 1982.
20. A. Matsumoto, *On improving the Boyer-Moore string matching algorithm* (October 1990). Personal communication.
21. R. Schaback, 'On the expected sublinearity of the Boyer-Moore algorithm,' *SIAM J. Comput.* **17,** (4), 648–658(1988).
22. W. Rytter, 'A correct preprocessing algorithm for Boyer-Moore string searching,' *SIAM J. Computing* **9,** 509–512(1980).
23. R. Giancarlo (December 1990). Personal communication.
24. J. J. Dongarra and E. Grosse, 'Distribution of mathematical software via electronic mail,' *Comm. ACM* **30,** (5), 403–407 (1987).
25. H. S. Bilofsky and C. Burks, 'The GenBank® genetic sequence data bank,' *Nucl. Acids Res.* **16,** 1861–1864(1988).