SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:815–833 (DOI: 10.1002/spe.367)

Fast string matching for multiple searches

Peter Fenwick*,†

Department of Computer Science, The University of Auckland, Private Bag 92019, Auckland, New Zealand

SUMMARY

We present a string matching or pattern matching method which is especially useful when a single block of text must be searched repeatedly for different patterns. The method combines linking the text according to digrams, searching on the least-frequent digram, and probing selected characters as a preliminary filter before full pattern comparison. Tests on real alphabetic data show that the number of character comparisons may be decreased by two orders of magnitude compared with Knuth–Morris–Pratt and similar searching, but with an initialization overhead comparable to five to ten conventional searches. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: string-searching; pattern matching; digram; linked list; intensive search; repeated searches

1. INTRODUCTION

One of the basic, and apparently simple, operations of text processing is searching for occurrences of a specified string. Most visibly, it occurs in text editors and word processors in connection with a 'Find' or 'Search' command. Less visibly, string searching and pattern matching is an essential component of many text compressors, especially those such as Ziv–Lempel 77 (LZ-77) that build explicit or implicit dictionaries of phrases and emit references to those phrases. All of these applications require fast and efficient pattern matching.

The general problem is to take a character sequence or string (the *text*) and search it for occurrences of another, shorter string (the *pattern*). There is an extensive literature on string searching and pattern matching algorithms; a recent book on the subject [1] has over 250 references. A survey recently posted to the Web by Charras and Lecroq [2] lists over 30 algorithms including code. Most string searching algorithms assume long text which may be searched a few times for fixed or stable patterns. This may be described as an *extensive* search and is the usual emphasis of papers in the literature. Another type

*Correspondence to: Peter Fenwick, Department of Computer Science, The University of Auckland, Private Bag 92019, Auckland, New Zealand.
†E-mail: p.fenwick@auckland.ac.nz

Copyright © 2001 John Wiley & Sons, Ltd.

Received 16 February 2000
Revised 10 July and 26 October 2000
Accepted 26 October 2000

```
for (ix = 1; ix < textLength; ix++)    // scan whole file
  if (patt0 == text[ix])               // one character match
    for (j = 1; j < pattLength; j++)   // full string comparison
      if (pattern[j] != text[j+ix])
        break;                         // mismatch
```

Figure 1. Essence of the simple or naive algorithm.

of pattern matching performs repeated searches of the text for many patterns; these may be described as *intensive* searches and are the main emphasis of this paper.

Another useful classification is that the 'traditional' algorithms take a given pattern and apply it to arbitrary text, while the techniques here are more useful for a given text which is scanned for arbitrary patterns.

2. EXTANT EXTENSIVE-SEARCH TECHNIQUES

In this discussion, and what follows, we assume two important strings of characters. The *text* string to be searched is an array `text[n]` of n characters, while the pattern `pattern[m]` is an array of m characters. The objective is to find all occurrences of `pattern` within `text`. An obvious method may be called the *naive* or *Brute Force* algorithm, broadly along the lines below and shown in Figure 1.

The two nested loops correspond to the two major phases of the algorithm; similar phases may be identified in most other algorithms.

1. The first (the outer loop) is a *search* along the text for a reasonable *candidate* string; in the naive algorithm this is as simple as searching for a matching first character.
2. The second phase (the inner loop) is a detailed comparison of the candidate against the pattern to *verify* the potential match.

The algorithm has a worst case time of $O(mn)$, because if `text` and `pattern` are all of the same character then every position during the search yields a possible match (n verifications), and every verification must proceed to completion (m comparisons per verification).

One problem with the naive algorithm comes from back-tracking or repeated examination of each character of text when patterns are partially matched. The classic paper by Knuth *et al.* [3] addresses this problem and shows that a preliminary analysis of the pattern allows the match to proceed with no back-tracking of the text. The overall time is linear in n and independent of m but in practice is little better than that of the naive algorithm. Their basic algorithm is shown in Figure 2 not including the code to set up the control array[‡].

[‡]Complete code for both Knuth–Morris–Pratt and Boyer–Moore–Horspool is given by Charas and Lecroq [2], including the generation of the control arrays.

Copyright © 2001 John Wiley & Sons, Ltd.

```
while (patIx < patLen)         // start of the KMP search
  {
  while(patIx > -1 && pattern[patIx] != text[txtIx])
    patIx = next[patIx];
  patIx++; txtIx++;
  if (txtIx > bytesInFile - patLen + 1)
      break;
  if (patIx >= patLen)                 // found a match
    {
    patIx = next[patIx];
    break;
    }
  } // end K M P search
```

Figure 2. Essence of Knuth–Morris–Pratt search.

The Knuth paper also discusses an algorithm by Boyer and Moore [4] which uses knowledge of the pattern to reduce the number of search comparisons, perhaps even as far as n/m. The Boyer–Moore (BM) algorithm does not itself address the back-tracking problem, but Knuth discusses the combination of BM scanning and Knuth–Morris–Pratt (KMP) verification. Unfortunately the theoretical benefits of the KMP algorithm are not realized in practice; back-tracking is seldom a problem and a KMP search is little better than the naive search for most files.

The BM method was extended by many authors. In particular Horspool [5] proposed the Boyer–Moore–Horspool (BMH) algorithm which is often regarded as the best general-purpose string searching algorithm. Briefly, it records the characters which are present in the pattern and tests whether a character of the text belongs to the pattern character set. If the test character does not occur in the pattern it is possible to skip forward by m characters (the pattern length) and repeat the examination. If the test character is one which occurs within the pattern, the examination indicates the probable alignment of the pattern with respect to the text to facilitate a full comparison to verify the match. The BMH algorithm therefore minimizes the search phase. It is shown in outline in Figure 3, again omitting the code to set up the control array.

Many of the published comparisons of the search techniques just count the number of character comparisons, often with artificial text. Although comparison counts are indeed adopted for much of this paper, their real value is extremely doubtful. Firstly, the true cost of a character comparison may be quite small in comparison with other overheads of the main scanning loop. If the loop includes 'helpful' array references, as in the KMP algorithm, those overheads may have a cost comparable with other parts of the loop. Optimization can also have major effects, whether it is performed by the compiler, or by cache operation and instruction rescheduling and overlap at program execution. Some of these points will be explored in a companion paper.

As an example of the 'traditional' search comparisons, Smit [6] compares the number of character comparisons for Naive (his 'Straightforward'), KMP and BM, using algorithms as originally published by the authors. On real text (an Afrikaans string of 5000 characters), he finds negligible difference in the number of actual character comparisons between Naive and KMP; both require from 1.1 to 1.2 comparisons for each text character, irrespective of the length of the search pattern. (This includes the

Copyright © 2001 John Wiley & Sons, Ltd.

```
while (patIx > 0)                 // loop until match is found
  {
  temp = txtIx;                   // last char of poss. match
  if (patIx >= 0)                 // still in pattern
    while (text[temp--] == pattern[patIx--]) // chars match
     if (patIx < 0)
              break;              // pattern matched
  if (patIx >= 0)                 // mismatch
    {
    txtIx += skip[text[txtIx]]; // advance text pointer
    patIx = pattLength-1;       // reset pattern pointer
    }
  } // end of matching loop
 if (patIx < 0)                    // we have a match
   {
   txtIx ++;                       // advance text pointer
   patIx = pattLength-1;           // reset pattern pointer
   }
```

Figure 3. Essence of the BMH search.

cost of pre-processing the pattern.) Excluding the cost of pre-processing, BM is always better than KMP, tending to an asymptote of 0.1–0.15 comparisons per text character for pattern lengths up to 15. With the preprocessing cost included, BM is less expensive for patterns longer than about four characters, with an asymptote of about 0.3. The results do not include actual execution times.

As an example of a 'real world' comparison, Hume and Sunday [7] do a very detailed analysis of the BM algorithm, producing a taxonomy of related algorithms and comparing the real performance on several different computers. It is one of the very few string searching papers to actually measure execution speeds, and on different computers. Their techniques include loop unrolling and a 'guard' test similar to the 'probe' introduced later in this paper. One of their better algorithms will be included in the tests of Section 7.

3. ZIV–LEMPEL COMPRESSION

The present work arose from experience with Ziv–Lempel compression [8,10] (usually known as LZ-77 and not to be confused with LZ-78 compression). LZ-77 compression processes the text serially by character and attempts to replace successive *phrases* of incoming text by references to earlier occurrences of like phrases within a *window* of the most recent characters. When a phrase has been processed the window advances to include the just-processed phrase and discards the older characters of the window. The LZ-77 implementation used by the author maintains the window as a character array of say the last 8 k characters. A *current pointer* marks the end of the window and advances cyclically around the array as the operation proceeds. The incoming data (the phrase to be matched) is read into a *lookahead area* in the array immediately after the current pointer. Simply advancing

Copyright © 2001 John Wiley & Sons, Ltd.

the current pointer will include the look-ahead area in the history window[§]. The matching operation involves searching back along the history window for the longest phrase or character string which matches that in the look-ahead area. It is usual to find short matches fairly quickly, but the search continues to older characters in the window in the hope of finding successively longer matches. When the best match is found, appropriate codes are emitted to describe the just-matched phrase and the boundary between the history buffer and look-ahead area is advanced to add the phrase to the history.

In comparison with pattern matching as addressed by Knuth *et al.* and Boyer *et al.*, this application requires repeated searching of slowly changing text for occurrences of rapidly changing patterns. It was described above as an *intensive* search.

4. GUTMANN PATTERN MATCHING

After an exhaustive study of pattern matching for LZ-77 compression, Gutmann [9] developed the following algorithm, which has not been published *per se*, but is described by Fenwick [10]. He found that complex and theoretically 'efficient' data structures were too often defeated by their generation, maintenance and traversal overheads, and that a simple structure with frequent but inexpensive operations was often preferable.

For his search algorithm the text buffer is a simple character array. Pairs of adjacent characters (or 'digrams') are hashed and used to index a table which defines links through the array, connecting character pairs of like hash values[¶]. An example of this table and the associated text is shown in Figure 4[‖]. When looking for a pattern we hash the first characters of the pattern and then trace along the chain corresponding to that hash value. The chain *must* include all occurrences of those first characters (plus any other pairs which collide in the hash table). As it stands, the hash table and links facilitate the *search* for likely candidates, but each candidate must still be *verified* as a valid match. In the simplest case the verification is a simple character by character comparison, but Gutmann devised a filter which greatly reduces the verification cost.

For this discussion, remember that we are working with a specialized form of string matching; we are looking not for matches to the current pattern, but for matches which *extend* the current pattern. Another match of the same length is of no interest whatsoever. Assume that at some stage we have a longest match of length λ, and are looking for a string of length $\lambda + 1$, as shown in Figure 5. As there is no benefit in considering any phrase with *length* $\leq \lambda$, we first look at the *probe_1* character in position $\lambda + 1$, one beyond the known best length. If that character does not match, the position cannot possibly give a longer match. If the *probe_1* character matches, we then compare the *probe_2* character at about the midpoint of the known best matching string. Only if the *probe_1* and *probe_2* comparisons succeed do we do a full character-by-character comparison to verify the match. If the match succeeds to length

[§]With this system the window size is reduced by the size of the look-ahead area; the effect on compression is negligible.

[¶]Rather than using digrams, Gutmann hashed several characters, the number selected according to string length and for best performance. The underlying principle though is identical to the description here.

[‖]This figure shows links from the earlier digrams to later ones, as is appropriate for normal pattern matching. For LZ-77 pattern matching it is better to have the links in the opposite direction, from the most recent digrams to the older ones.

Copyright © 2001 John Wiley & Sons, Ltd.

Input Text		
Index	Character	link
1	M	13
2	i	14
3	s	6
4	s	7
5	i	14
6	s	15
7	s	○
8	i	○
9	p	○
10	p	○
11	i	○
12	-	○
13	M	○
14	i	○
15	s	○
16	s	○

Digram table	
digram	head
-M	12
i-	11
ip	8
is	2
Mi	1
pi	10
pp	9
si	4
ss	3

Figure 4. Input text array and digram table, with links.

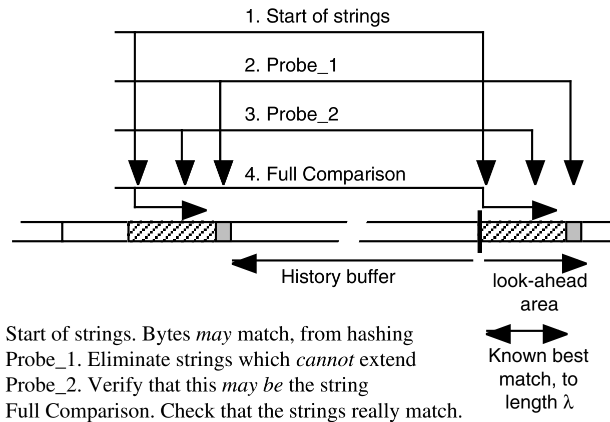

Figure 5. Gutmann string matching.

$\lambda + 1$, we then extend the match as far as possible. Methods also involving probes were developed by both Raita [12] and by Hume and Sunday [7] for the BMH algorithm.

Testing a given candidate string then requires only the following steps, most of which are simple and fast.

1. Check that this position is still valid (not beyond the end of the hash chain).

2. Compare the *probe_1* characters.
3. Compare the *probe_2* characters.
4. Do a full comparison of the two strings, with possible extension.
5. Step to the next position in the hash chain.

In this algorithm the linked lists of like digrams facilitate the search for likely candidates, sometimes reducing them to 1% or less of the text length. Gutmann's contribution is the use of the two probes as an initial filter to reduce the cost of the final verification. His experiments showed that about 50% of possible phrases were eliminated on the first probe, and that fewer than 10% survived the second probe as well and needed a full comparison.

We therefore have an algorithm which accelerates both *searching*, using lists based on the digrams, and *verification*, by probing judiciously selected characters of the text.

The underlying principle is that adjacent characters in a text string tend to be highly correlated; if we have a match to one character it is likely that the adjoining characters will also match. A much more sensitive test is on well-separated characters. Here the *probe_1* characters are ones with low correlation with the leading characters on which the hash chain is based. The *probe_2* test just selects the character which probably has the lowest correlation with the start of the string (which probably matches because of the list to which it belongs) and the *probe_1* character (which could extend the string). Gutmann suggested that a further refinement might be to select as the *probe_2* character one with a low probability.

5. THE NEW SEARCH ALGORITHM

The new algorithm is based on the Gutmann algorithm, but with two changes. The first change is an adaptation for 'conventional' searching, as perhaps in an editor, with forward searching and static patterns. The second and crucial change concerns the list that is traversed to find likely candidates. Gutmann used the list for the two characters at the start of the look-ahead area, as an obvious choice for LZ-77 compression which extends the test string during compression. With a fixed-length pattern, we can select *any* of its internal digrams as ones to search for**. The least-frequent digram has the shortest associated list and choosing it minimizes the search length and the number of candidates to verify.

The first step is to read in the entire input text, linking and counting digrams, using code as in Figure 6. This builds up a table similar to that shown in Figure 4 but with digram counts added.

Then for each search the following steps are taken.

1. Find the least probable digram in the pattern (the digram with the lowest frequency). This becomes a *pivot* during the search.
2. Choose two other test or probe positions. If the pivot is near the middle of the pattern the probe positions should be at the two ends of the pattern. If the pattern is at one end they should be near the other end and the middle. We try to find two test positions which are remote from and relatively uncorrelated with the characters of the pivot.

**At the cost of slightly more work as the match extends, we could change the pivot digram in LZ-77 searching as well.

Copyright © 2001 John Wiley & Sons, Ltd.

```
for (txtIx = 1; txtIx < bytesInFile; txtIx++)
  {
  hashIx = hashText(text, txtIx); // call the hash function
  if(hashCount[hashIx] == 0)
    hashFirst[hashIx] = txtIx;  // link to start of list
  else
    {
    prev = hashLast[hashIx];    // previous end of list
    txtLink[prev] = txtIx;      // link prev to this
    }
  hashLast[hashIx] = txtIx;     // current end of list
  txtLink[txtIx] = 0;           // mark end of list
  hashCount[hashIx]++;          // another entry
  }
```

Figure 6. Creation of the digram links.

```
// probe1 and prob2 are the actual characters to compare
// cmpStart and cmpStop are the string compare limits, having
// regard to positions tested already by probe1 and probe2.
//
while (testPosn > 0)                 // link ends with -1
  {
  testStart = testPosn - pivot;      // offset of comparison
  if (probe1 == text[testStart+test1])
    if (probe2 == text[testStart+test2])
      for (i = cmpStart; i <= cmpStop; i++) // full str. compare
        if (pattern[i] != text[testStart+i])
             // ....  mismatch
  testPosn = txtLink[testPosn];       // link to next target
  }
```

Figure 7. Essence of final digram search.

3. Compare the two probe positions, remotest first; skip the next step if either test fails.
4. Do a full comparison of the two strings, reporting success if there is a total match. Test characters at the ends of the pattern are excluded from this comparison if they have been ‘probed’ already.
5. Step to the next position in the hash chain if there is no match or more matches are needed.

The pattern comparison procedure is now rather more complex, because the pivot may occur anywhere within the pattern. With patterns of four or more characters there are the three conditions illustrated in Figure 8. Each square represents a pattern character, with ‘P’ the two pivots. With the pivot at one end the initial probe (position ‘1’) is at the other end and the second probe is near the middle of the pattern. With the pivot in the middle of the pattern the situation can be much more complex, especially with a long pattern (which may resemble one of the other two cases). For simplicity we just probe at the two ends of the pattern.

Copyright © 2001 John Wiley & Sons, Ltd.

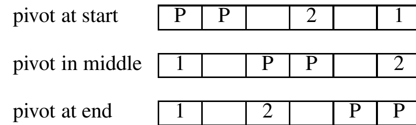

Figure 8. Test character and pivot positions.

Patterns shorter than four characters need special treatment (and single characters cannot be handled at all by the digram structure and need a serial search). Patterns of two characters can be compared immediately to confirm that they are not a pair that collides during the hashing. Patterns of three characters have a single character either before or after the pivot. This non-pivot character should be tested first, before the pivot characters are confirmed.

6. THE TEST PROCEDURE

Several different string search algorithms were programmed and compared by reporting the number of character comparisons. The search algorithms were as follows.

Naive is the simplest algorithm of all, as given earlier. It is representative of most of the extensive search algorithms and has costs similar to that of KMP for most text.

Knuth–Morris–Pratt is the standard KMP algorithm.

Boyer–Moore–Horspool is chosen as a good example of the best extensive search algorithms.

Tuned Boyer–Moore is Hume and Sunday's revision of the BM algorithm. It resembles Horspool's algorithm, but makes vigorous efforts to achieve efficient code with loop unrolling and with pointers preferred to subscripted array references.

Init digram, 0 probes includes the digram table and the system of lists which link digrams of like hash value. The linked digram is always the first one of the pattern. It is the method which Gutmann sought to improve by adding the probes. One minor optimization is that the verification is done backwards so that less-correlated characters are compared first.

Init digram, 2 probes is Gutmann's method with links of the initial digrams and two character probes as a preliminary filter before verifying each digram on the list.

Best digram, 2 probes is the revised Gutmann search procedure, with the least frequent digram selected as the pivot and search list.

Copyright © 2001 John Wiley & Sons, Ltd.

7. THE TEST RESULTS

The results here use two text files from the Calgary compression corpus:

Book1. ‘Far From the Madding Crowd’, Thomas Hardy, 776 781 characters; and

Book2. ‘Principles of Computer Speech’, Ian Witten, 610 856 characters.

About 30 words from a paragraph near the start of each text were used as test patterns for each file, with the hash function a concatenation of the six least-significant bits of the two characters.

Table I shows results for four words from Book1. The ‘naive’ method requires a few more character comparisons than the file size. It is useful to confirm that the more efficient methods have found all occurrences of the pattern, but is otherwise unimportant. All of the test patterns require about 800 000 character comparisons, with any changes largely dependent on the probability of the initial letter.

Considering the details for the word ‘weakness’ we have the following comparisons between the algorithms.

Naive. Relatively few of the text characters match the initial pattern character and require further comparisons (14 071 of 761 778, or 1.8%). Few of these full comparisons proceed beyond the first test or second character of the pattern. (There are only 2044 comparisons beyond the first.)

Knuth–Morris–Pratt. The KMP algorithm actually saves very few comparisons compared with the simpler naive method , showing that the back-tracking is for this text very much a ‘non-problem’.

Boyer–Moore–Horspool. Because the BMH algorithm uses knowledge of the pattern it needs to examine only about 15% of the text characters and only about 7000 of these warrant a further examination.

Tuned Boyer–Moore. This ‘improved’ algorithm requires more character comparisons than BMH, because the loop unrolling means that a success is sometimes followed by several unnecessary tests. We see later that because of its efficient design, the increased number of comparisons is more than offset by reduced overheads in other areas.

Initial digram, 0 probes. The low frequency of the initial digram ‘we’ means that only about 0.25% of the whole text needs to be examined, reducing the number of character comparisons by about 135 times relative to naive or KMP, or 20 times relative to BMH.

Initial digram, 2 probes. The important point here is that having identified a likely candidate, we next test a character remote from the initial digram and then yet another, separated one. Here 96 candidates survive the first probe, and eight survive the second, leaving only 47 character comparisons within strings. Using the character probes has reduced the number of comparisons to about 35% of that without probes.

Best digram, 2 probes. In comparison with the previous case, choosing the rarer ‘kn’ digram instead of the initial ‘we’ reduces the number of candidate strings, before probing, by about 65%. The number of strings which need a full comparison is reduced only from eight to six; the improvement is in the faster elimination of near-matches.

Copyright © 2001 John Wiley & Sons, Ltd.

Table I. Results from Book1, text size = 761 778 characters.

	probe_1 or skips	probe_2	Strings tested	Final char compares	Total char compares
Searching for ‘carried’, 33 occurrences					
Naive (Brute Force)			12 685	14 419	783 189
Knuth–Morris–Pratt					781 418
Boyer–Moore–Horspool	130 176				137 725
Tuned Boyer–Moore	140 048				150 375
Init digram, no probes	1385		1385	4504	4504
Init digram, 2 probes	1385	46	33	198	1629
Best digram, 2 probes	659	105	33	165	929
Searching for ‘damp’, seven occurrences					
Naive (Brute Force)			26 623	27 151	795 921
Knuth–Morris–Pratt					795 385
Boyer–Moore–Horspool	203 235				206 275
Tuned Boyer–Moore	208 634				214 331
Init digram, no probes	537		537	1602	1602
Init digram, 2 probes	537	7	7	21	565
Best digram, 2 probes	537	7	7	21	565
Searching for ‘their’, 241 occurrences					
Naive (Brute Force)			50 027	75 848	844 618
Knuth–Morris–Pratt					818 554
Boyer–Moore–Horspool	174 788				183 556
Tuned Boyer–Moore	189 006				198 992
Init digram, no probes	15 995		15 995	57 811	57 811
Init digram, 2 probes	15 995	675	623	1346	18 016
Best digram, 2 probes	816	249	241	723	1788
Searching for ‘weakness’, six occurrences					
Naive (Brute Force)			14 071	16 115	784 885
Knuth–Morris–Pratt					782 830
Boyer–Moore–Horspool	115 977				123 113
Tuned Boyer–Moore	127 258				133 073
Init digram, no probes	1904		1904	5852	5852
Init digram, 2 probes	1904	96	8	47	2047
Best digram, 2 probes	668	25	6	36	729

Table II shows the digram frequencies for the words used in Table I. For ‘damp’, the first digram is the least frequent and the revised algorithm shows no improvement. With ‘their’, ‘ei’ has a frequency about 5% that of the leading ‘th’, with the search distance reduced accordingly. With ‘weakness’, the number of comparisons is reduced by 1904/668, or about 2.85 times.

In the second case, for the word ‘their’, moving from the relatively frequent ‘th’ to the less frequent ‘ei’ reduces the comparisons by a factor of about 100. For the last word ‘weakness’, the list for ‘we’

Copyright © 2001 John Wiley & Sons, Ltd.

Table II. Book1 digram frequencies.

carried		damp		their		weakness	
ca	1385	da	537	th	15 995	we	1904
ar	4259	am	1264	he	17 470	ea	3833
rr	659	mp	656	ei	816	ak	1053
ri	2544			ir	1410	kn	668
ie	1524					ne	3420
ed	6207					es	4176
						ss	1938
Ratios of initial to lowest digram counts							
2.10		1.00		19.6		2.85	

Table III. Full results for Book1 and Book2.

	Candidates or skips	probe_2	Strings	String char compare	Total char compares
(a) Results for Book1 (finding 31 words).					
Naive (Brute Force)	742 581			847 068	24 678 938
Knuth–Morris–Pratt					24 564 952
Boyer–Moore–Horspool	5 739 427				6 066 659
Tuned Boyer–Moore	6 207 692				6 514 154
Init digram, no probes	88 937		88 937	282 038	282 038
Init digram, 2 probes	88 937	10 429	9124	21 334	120 700
Best digram, 2 probes	34 120	9657	8754	20 152	63 929
(b) Results for Book2 (finding 32 words).					
Naive (Brute Force)	836 114			947 864	20 495 224
Knuth–Morris–Pratt					20 370 190
Boyer–Moore–Horspool	4 132 127				4 437 696
Tuned Boyer–Moore	4 541 636				4 800 713
Init digram, no probes	98 169		98 169	307 173	307 173
Init digram, 2 probes	98 169	15 596	12 782	32 169	145 934
Best digram, 2 probes	35 051	15 026	12 758	30 478	80 555

has 1904 members; the first probe of the Gutmann algorithm eliminates 95% of those and 87% of the remainder are eliminated by the second probe. The revised algorithm uses the list for ‘kn’ (see Table II) which has only 668 members and a corresponding reduction in the number of probe_1 comparisons. Only six full string comparisons are needed, which is a considerable difference from the 1904 needed by the algorithm without probes.

These results are confirmed by the results of Tables III(a) and (b) which give the combined results for both of the test files. (Both involve about 30 searches for different words, or a total text search over

Copyright © 2001 John Wiley & Sons, Ltd.

Table IV. Effect of hash function changes.

Hash function	Book1 count	Book2 count	Book1 change	Book2 change
4 bits	106 451	113 825	66.5%	41.3%
5 bits	66 569	82 359	4.1%	2.2%
6 bits	63 929	80 555	0.0%	0.0%
7 bits	63 360	77 814	−0.9%	−3.4%
Random table	75 849	96 502	18.6%	19.8%

nearly 23 million characters.) The improvement in character comparisons with respect to naive is 385 times for Book1 and 254 for Book2.

8. HASH FUNCTION GENERATION

All of the work above was done with a rudimentary hash function, just the concatenation of the six least significant bits of each character. It is appropriate to try the effect of changing the numbers of bits used and also changing to a better hash function which indexes into a table of random numbers.

The results are shown in Table IV, showing the number of character comparisons for the final digram algorithm. The two right-hand columns show the change from the '6-bit' hash used so far; a decrease is an improvement. There is little real change in moving from 6 bits to 5 or 7 bits. Even just 5 bits captures the essence of the alphabet and symbol frequencies. Book2 is a technical book, written in a mark-up language with frequent control codes. These codes are separated out with the 7-bit codes and the performance improves. Using only 4 bits of each character is an extreme which should not be considered (although the absolute performance is still excellent when compared with other algorithms).

The surprise is that using a 'good' randomizing hash function degrades the performance by about 20%. This is because the good hash function is simply too good at distributing values towards an even distribution and destroys the variation in hash frequency which underlies the operation of the new algorithm. We need a function which preserves the uneven digram distribution.

We can also consider the effects of using n-grams in place of the digrams used so far. The hash table has 4096 entries and with 6-bit digrams can accommodate an alphabet of 64 symbols which is adequate for normal text. Expanding to trigrams or longer needs either a much larger table (most obviously $2^{18} = 262\,144$ entries, or a better designed hash function. While it is certainly essential to avoid collisions between frequent trigrams, collisions between infrequent trigrams probably matter less. Overall it is essential to maintain the identity of at least the more frequent trigrams.

For smaller alphabets, the hash table must be kept reasonably full to maximise the number of active chains and minimize the average length of each chain. Trigrams are the obvious choice with 4-bit characters (such as hexadecimal digits) and guarantee collision-free operation with the 4096 entry table. For DNA with an alphabet of only four symbols, a concatenation of 6-grams is appropriate with

Copyright © 2001 John Wiley & Sons, Ltd.

Table V. Full results for DNA search (207 363 symbol text, 30 patterns).

Line		Candidates or skips	probe_2	Strings	String char compare	Total char compares
1	Naive (Brute Force)	1 657 565			2 275 505	8 496 365
2	Knuth–Morris–Pratt					7 270 239
3	Boyer–Moore–Horspool	1 667 437				2 458 435
4	Tuned Boyer–Moore	2 520 411				1 143 480
5	Initial digram 0 probes	444 195		444 195	1 506 329	1 506 329
6	Initial digram 2 probes	444 195	116 098	32 115	44 862	605 155
7	Best digram 2 probes	208 979	54 617	13 798	22 043	285 639
8	Best 6-gram 2 probes	851	278	71	465	1594
9	Initial 6-gram 2 probes	2821	760	211	552	4133

a 4096 entry table. It can (or should) guarantee no hash collisions at all and ensure that searching is performed only on unique 6-gram prefixes[††].

9. SEARCH OF A DNA FILE

As an example of searching with a very small alphabet, we use as text a sequence of 207 363 DNA base pairs, with results shown in Table V. The patterns are 30 randomly selected DNA sequences, with lengths uniformly distributed in the range 30 to 90. Searching uses the algorithms listed in Table III, but adding a digram algorithm modified to use a 6-character hash designed for DNA.

To a first approximation we can assume that the DNA symbols are equiprobable so that the naive algorithm will match on 25% of the text characters for each of the 30 test patterns. The predicted 155 000 matches compares well with the observed 165 000. Similarly the BMH algorithm will have to examine every fourth character. The Tuned BM (TBM) however has a three-fold loop unrolling; as it too must examine each fourth text character we can expect frequent overruns and unnecesary testing, raising its number of tests by about 27% compared with BMH. (The algorithm optimization is clearly inappropriate to the very small DNA alphabet and should be reworked for this case.)

With 16 possible digrams, each digram list should have $207\,363/16 \approx 13\,000$ members, or about 390 000 for the full 30 searches. This is in accord with the observed 441 000 initial tests. Each character probe eliminates about 75% of the cases (as expected from four equiprobable symbols). Line 7 of Table V shows that the text is perhaps not quite as random as supposed; choosing the right digram can still halve the number of searches. Again each probe eliminates about 75% of the candidates.

[††] The obvious solution of taking each DNA character code mod 4 is unsuitable because both C and G yield three.

Copyright © 2001 John Wiley & Sons, Ltd.

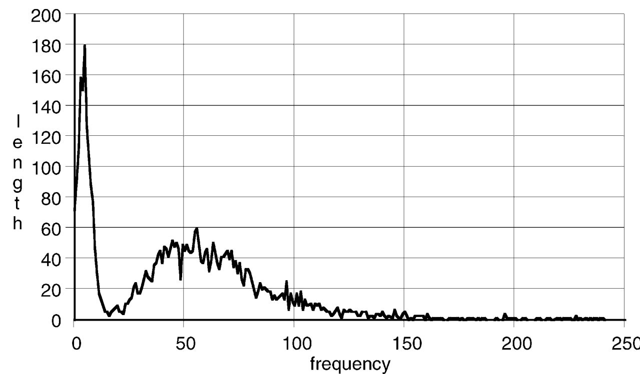

Figure 9. Distribution of hashed link lengths for DNA data.

Line 8 shows the new search algorithm tuned to DNA, using 6-grams to form the 'hash' value, although with six sets of two bits being combined into 12 bits, the hashing is completely collision-free. We would expect each hash list to contain $207\,363/4096 = 51$ entries, giving 1530 candidates for the 30 patterns. In fact, selecting the best 6-gram cuts this value by half, in line with what was seen for the previous line of the table. Each probe again eliminates 25–30% of the strings, leaving only 71 for further examination. As this is over all 30 patterns, only about two patterns on average survive the two probes and need to even start a full string comparison. (Except for one outlier with 14 candidates surviving the two probes, the average was only two survivors, with eight having none and seven only one.) The result is that the number of character comparisons has been reduced to $1/4500$ of that needed for KMP, or about $1/700$ of TBM.

A closer examination shows that it may be unnecessary to select the best 6-gram and that using the initial 6-gram should give adequate performance. The overheads of preprocessing the complete DNA pattern (which tends to be much longer than a typical text pattern) are considerable and certainly outweigh the benefits of reducing the number of candidate patterns. This case, for an initial 6-gram with two probes and no selection of the best is given in line 9 of Table V; it is the 6-gram analogue of line 6. Compared with line 8, it involves about three times as many examinations before the final full comparison. The overall number of comparisons is about 2.6 times that of line 8, but is still far less than with any of the older methods.

Although DNA is known from compression results to have an entropy very close to 2 bits per base pair, this certainly does not imply equiprobable symbols. Figure 9 shows the distribution of link lengths for the DNA file for 6-gram hashing, omitting a few outliers which do not show on the diagram. (The longest chain had 637 entries.) It is clear that the distribution is far from uniform, which in turn implies that the frequency-sensitive algorithm as given here is useful. This is confirmed by the results.

But whether an algorithm of this sort is indeed useful for DNA is quite another question. DNA contains a large amount of 'junk' or noise and most DNA matching programs must handle large

Copyright © 2001 John Wiley & Sons, Ltd.

amounts of inserted (or even deleted) material. Pattern matching of that sort is a completely different problem from that discussed here.

10. OVERHEADS AND COSTS

The first and obvious overhead is the space for the digram hash table and the digram links. Each entry for both is either an index into the text or a pointer into the text; a size of 32 bits or 4 bytes is appropriate for most computers. Added to this is the digram count for each entry, either 2 or 4 bytes. A hash table of 4 096 has been used and seems appropriate, giving an overhead of 24 or 32 kbytes, depending on the size of the associated counts. The links add 4 bytes per character, giving a total of 5 bytes per text character, to whatever number of characters is needed to hold the text.

One of the big problems in a study such as this is comparing the speed (or other performance measure) of different algorithms in a manner which is independent of the computer. The measure which has been used here is the number of character comparisons. Each comparison needs two accesses to character arrays and might be expected to dominate the execution time if we assume that the other overheads are comparable between algorithms. But this is a questionable assumption. The KMP and BMH algorithms also require housekeeping array references. As array references, whether character or integer, may be relatively expensive, the housekeeping costs too should be counted towards a performance measure. We ignore cache locality, whether temporal or spatial, as being too complex in a simple discussion.

When searching for a pattern of m characters in text of n characters, some examples are as follows.

Naive. This needs two array references for each unsuccessful comparison, probably $2n$ array references in total if we ignore the extra costs following a match. (There is only one array reference per position, or n in total with the obvious optimization of reading the initial pattern character into a local variable.) There is no set-up cost.

KMP. This examines every character position and also needs $2n$ array references. However, each unsuccessful comparison requires an additional reference to the state machine and possibly a reexamination of the mismatched character. We can therefore ascribe a cost of $3n$ references. Initialization involves one pass through the pattern.

BMH. Horspool's algorithm (BMH) makes n/m comparisons ($2n/m$ character array references) in the basic scan. However, each mismatch requires two more references to determine the pattern advance or realignment (and most comparisons fail). We therefore get a basic cost of about $4n/m$ memory references for the fastest text scan. Initialization for BMH involves a scan through the pattern. But as well as this cost proportional to the pattern length, it also requires clearing an array equal in size to the alphabet. Even with block fills or replaces (rare on modern computers) this is an expensive operation.

TBM. Hume and Sunday's 'Tuned Boyer–Moore' resembles Horspool's algorithm, but makes vigorous efforts to achieve efficient code. We can expect a similar number of array references, but many of them are in unrolled loops with minimal overhead. Whereas the other test cases

Copyright © 2001 John Wiley & Sons, Ltd.

Table VI. Timings for various activities.

Activity	Time (ms)	Digram link cost
Reading file 'Book1'	100.00	—
Creating links	61.2500 ± 0.3400	—
Naive ('Brute force')	15.5460 ± 0.0070	3.9
Knuth–Morris–Pratt	45.8440 ± 0.0260	1.3
Boyer–Moore–Horspool	11.1960 ± 0.0100	5.5
Tuned Boyer–Moore	4.0910 ± 0.0030	15.0
Digram	0.1934 ± 0.0007	317.0

use traditional array subscripting, TBM relies on pointer manipulation to further increase the execution speed‡‡.

Again, some algorithms (not discussed here) test less frequent characters before more frequent ones and develop mapping tables to perform the necessary test ordering. Using these tables adds further expensive array accesses; the benefits of optimized character testing may have to be offset against the added cost of indexing through the mapping tables.

Apart from the character comparisons which are easily counted as two array references each, the final digram algorithm has only one other array reference (in traversing the digram list) when examining each candidate. It does however have extra overheads in setting up the structure of digram lists. In the test implementation these add ten array references to the reading of each character, though several could be avoided by careful coding. The ten added array references must be compared with the two, three or four references per character for some other algorithms. Remember too that the invisible overheads of file reading can be significant on some computers.

Overall, we estimate that the setting up overhead of the three 'digram' algorithms is equivalent to five or six searches on the simpler algorithms, or ten or 12 searches with BMH. For a few searches of the same text the setting up time of the new algorithms might not be justified, but for more than a dozen searches they should be much faster overall.

Finally, we measured the actual times for some of the activities and searches. The results are shown in Table VI, for a Macintosh G4 computer with a 500 MHz PowerPC processor. The compiler was MetroWerks version 4.0, at the highest level of optimization. Included in this table is the time for Hume and Sunday's 'TBM' algorithm [7], using code from their web site. A major problem with timing was that the combination of a fast computer and relatively slow clock (about eight million processor clocks per system tick) made many of the algorithms execute in much less than one clock tick, even on a

‡‡The source code is available by sending e-mail to `netlib@research.att.com`, with the message body '`send index bmsrc from stringsearch`', and extracting the file `uf.fwdg.md2.c`.. See the original paper for details of the file contents and other files from the `stringsearch` suite.

Copyright © 2001 John Wiley & Sons, Ltd.

0.75 Mbyte file. Eventually it was necessary to repeat each search of each word until at least 100 ticks had elapsed and then report the average time for each search.

The times are the averages for finding each of the 31 test words for ‘Book1’ used in earlier tests. Each measurement was repeated 10 times to allow an estimate of both average and standard deviation of each time. Table VI also shows the cost of setting up the digram search links, relative to the cost of each search algorithm. All of the search times include the overheads of preprocessing the pattern to compute control tables, optimum digrams and so on. The cost of setting up the links is seen to be equivalent 3.9 naive searches, 1.3 KMP searches, 5.5 BMH searches, or 15 TBM searches, in general agreement with the earlier predictions.

11. NOTES ON TIMINGS AND COMPARISONS

This paper has followed the lead of many, and indeed most, authors in comparing algorithms only from the relative number of comparisons. As some referees pointed out, this is an entirely misleading metric, because times are usually dominated by other aspects of the algorithm. For example an algorithm which needs frequent references to a control array, such as KMP, must have those references added into the cost of the algorithm. A critical analysis of an algorithm, as done by Hume and Sunday for the BM search, can often get a marked improvement. But as Hume and Sunday also point out, many of these algorithms have some very subtle difficulties; they state that of four implementations of one version of BM, they found only one correct.

The author certainly knew of these matters, but realized their true importance only after this paper had been submitted, when the algorithms were tried on several other computers, and gave quite different ratios of relative speeds. Investigation of the *real* performance grew into a second paper which has been submitted separately and emphasizes some pitfalls of algorithm comparison [11]. It shows that naive comparisons, for example from character comparisons, are almost worthless in the real world. It also compares algorithms on different computers and shows that relative speeds on one seldom transfer well to another.

12. CONCLUSIONS

This paper describes a new string searching or pattern matching algorithm with very high performance. In contrast to most other algorithms, it involves preprocessing the text, with an overhead measured in character comparisons of about half a dozen KMP searches, or about 10 or 12 BMH searches. Tests on the text of two books shows that the actual number of character comparisons is improved by up to 400 times over simple searches, or up to 80 times over BMH searches.

Because it involves significant preprocessing of the text to be searched the new algorithm is best where repeated searches of a single text are required (most search algorithms process only the pattern to be found). It is not appropriate where the text is searched only a few times.

Tests on a 200 kbyte file of DNA show that exact matching may require only 0.1% of the character comparisons required by standard pattern matching algorithms.

Copyright © 2001 John Wiley & Sons, Ltd.

Timing tests show that preprocessing is equivalent to from 1.3 KMP searches to 12 searches with TBM, Hume and Sunday's implementation of BM. Once that processing is complete, searching then proceeds at about 250 times as fast as KMP or 30 times as fast as TBM.

ACKNOWLEDGEMENTS

The author thanks the University of Auckland for provision of research facilities. Special thanks are due to Peter Gutmann who initially provoked the author's interest in this topic and who developed the first stage of improvements leading to the new algorithm. Thanks are due to the referees who suggested many improvements, especially pointing out the paper by Smit, and the paper and algorithms by Hume and Sunday.

REFERENCES

1. Stephen GA. *String Searching Algorithms*, World Scientific: Singapore, 1994.
2. Charras C, Lecroq T. *Exact String Matching Algorithms*. http://www-igm.univ-mlv.fr/lecroq/string/ [1997].
3. Knuth DE, Morris JH Jr, Pratt VR. Fast pattern matching in strings. *SIAM Journal of Computing* 1977; **6**2:323–350.
4. Boyer RS, Moore JS. A fast string searching algorithm. *Communications of the ACM* 1977; **20**(10):762–772.
5. Horspool RN. Fast string searching. *Software—Practice and Experience* 1991; **21**(11):1221–1248.
6. Smit G de V. A comparison of three string matching algorithms. *Software—Practice and Experience* 1982; **12**:57–66.
7. Hume A, Sunday D. Practical fast searching in strings. *Software—Practice and Experience* 1980; **10**(6):501–506.
8. Ziv J, LempeL A. A universal algorithm for universal data compression. *IEEE Transactions on Information Theory* 1977; **IT-23**(3):337–343.
9. Gutmann PC. Personal communication.
10. Fenwick PM. Differential Ziv–Lempel text compression. *J-UCS (Journal of Universal Computer Science)* 1995; **1**(8):587–598.
11. Fenwick P. Some perils of performance prediction: a case study on pattern matching. *Software—Practice and Experience* 2001; **31**(9):835–843.
12. Raita T. Tuning the Boyer–Moore–Horspool string searching algorithm. *Software—Practice and Experience* 1992; **22**(10):879–884.

Copyright © 2001 John Wiley & Sons, Ltd.