A FAST STRING SEARCHING ALGORITHM

BY ROBERT S. BOYER AND J STROTHER MOORE

XEROX
PALO ALTO RESEARCH CENTER

A FAST STRING SEARCHING ALGORITHM

BY ROBERT S. BOYER* AND J STROTHER MOORE**

CSL-76-1 JULY, 1976

An Algorithm is presented that searches for the location, **i**, of the first occurrence of a character string, **pat**, in another string, **string**. During the search operation, the characters of **pat** are matched starting with the last character of **pat**. The information gained by starting the match at the end of the pattern often allows the algorithm to proceed in large jumps through the text being searched. Thus, the algorithm has the unusual property that, in most cases, not all of the first **i** characters of **string** are inspected. The number of characters actually inspected (on the average) decreases as a function of the length of **pat**. For a random English pattern of length 5, the algorithm will typically inspect **i/4** characters of **string** before finding a match at **i**. Furthermore, the algorithm has been implemented so that (on the average) fewer than **i+patlen** machine instructions are executed. These conclusions are supported with empirical evidence and a theoretical analysis of the average behavior of the algorithm. The worst case behavior of the algorithm is linear in **i+patlen**, assuming the availability of array space for tables linear in **patlen** plus the size of the alphabet.

KEY WORDS

bibliographic search, computational complexity, information retrieval, linear time bound, pattern matching, text-editing

CR CATEGORIES

3.74, 4.40, 5.25

----

*Computer Science Group, Stanford Research Institute, Menlo Park, Ca. 94025. This work was partially supported by ONR Contract NOOO14-75-C-0816.

**Formerly at the Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, Ca 94304, currently at Computer Science Group, Stanford Research Institute, Menlo Park, Ca 94025.

Introduction

Suppose that **pat** is a string of length **patlen** and we wish to find the position, **i**, of the leftmost character in the first occurrence of **pat** in some string **string**:

```
pat:          AT-THAT
string:   ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
i:                                  ↑
```

The obvious search algorithm considers each character position of **string** and determines whether the successive **patlen** characters of **string** starting at that position match the successive **patlen** characters of **pat**. Knuth, Morris, and Pratt [4] have observed that this algorithm is quadratic. That is, in the worst case, the number of comparisons is on the order of **i*patlen**[1].

Knuth, Morris, and Pratt have described a linear search algorithm which preprocesses **pat** in time linear in **patlen** and then searches **string** in time linear in **i+patlen**. In particular, their algorithm inspects each of the first **i+patlen-1** characters of **string** precisely once.

We now present a search algorithm which is usually "sublinear": it may not inspect each of the first **i+patlen-1** characters of **string**. By "usually sublinear" we mean that the expected value of the number of inspected characters in **string** is **c*(i+patlen)**, where **c** < 1 and gets smaller as **patlen** increases. There are patterns and strings for which worse behavior

----
[1]The quadratic nature of this algorithm appears when initial substrings of **pat** occur often in **string**. Because this is a relatively rare phenomenon in string searches over English text, this simple algorithm is *practically* linear in **i+patlen** and therefore acceptable for most applications.

is exhibited. However, Knuth, in [5], has shown that the algorithm is linear even in the worst case.

The actual number of characters inspected depends on statistical properties of the characters in **pat** and **string**. However, since the number of characters inspected, on the average, decreases as **patlen** increases, our algorithm actually speeds up on longer patterns.

Furthermore, the algorithm is sublinear in another sense: it has been implemented so that on the average it requires the execution of fewer than **i+patlen** machine instructions per search.

The organization of this paper is as follows: In the next two sections we give an informal description of the algorithm and show an example of how it works. We then define the algorithm precisely and discuss its efficient implementation. After this discussion we present the results of a thorough test of a particular machine code implementation of our algorithm. We compare these results to similar results for the Knuth, Morris, and Pratt algorithm and the simple search algorithm. Following this empirical evidence is a theoretical analysis which accurately predicts the performance measured. Next we describe some situations in which it may not be advantageous to use our algorithm. We conclude with a discussion of the history of our algorithm.

An Informal Description

The basic idea behind the algorithm is that more information is gained by matching the pattern from the right than from the left. Imagine that **pat** is placed on top of the lefthand end of **string** so that the first characters of the two strings are aligned. Consider what we learn if we fetch the **patlen**th character, **char**, of **string**. This is the character which is aligned with the *last* character of **pat**.

Observation 1: If **char** is known not to occur in **pat**, then we know we need not consider the possibility of an occurrence of **pat** starting at **string** positions 1, 2, ... or **patlen**: Such an occurrence would require that **char** be a character of **pat**.

Observation 2: More generally, if the last (rightmost) occurrence of **char** in **pat** is $\mathbf{delta}_1$ characters from the right end of **pat**, then we know we can slide **pat** down $\mathbf{delta}_1$ positions without checking for matches. The reason is that if we were to move **pat** by less than $\mathbf{delta}_1$, the occurrence of **char** in **string** would be aligned with some character it could not possibly match: Such a match would require an occurrence of **char** in **pat** to the right of the rightmost.

Therefore, unless **char** matches the last character of **pat** we can move past $\mathbf{delta}_1$ characters of **string** without looking at the characters skipped. $\mathbf{delta}_1$ is a function of the character **char** obtained from **string**. If **char** does not occur in **pat**, $\mathbf{delta}_1$ is **patlen**. If **char** does occur in **pat**, $\mathbf{delta}_1$ is the difference between **patlen** and the position of the rightmost occurrence of **char** in **pat**.

Now suppose that **char** matches the last character of **pat**. Then we must determine whether the previous character in **string** matches the 2nd from the last character in **pat**. If so, we continue backing up until we have matched all of **pat** (and thus have succeeded in finding a match), or else we come to a mismatch at some new **char** after matching the last **m** characters of **pat**.

In this latter case, we wish to shift **pat** down to consider the next plausible juxtaposition. Of course, we would like to shift it as far down as possible.

Observation 3-a: We can use the same reasoning described above -- based on the mismatched character, **char**, and $\mathbf{delta_1}$ -- to slide **pat** down **k** so as to align the two known occurrences of **char**. Then we will want to inspect the character of **string** aligned with the last character of **pat**. Thus, we will actually shift our attention down **string** by **k+m**. The distance, **k**, we should slide **pat** depends on where **char** occurs in **pat**. If the rightmost occurrence of **char** in **pat** is to the right of the mismatched character (i.e., within that part of **pat** we have already passed) we would have to move **pat** backwards to align the two known occurrences of **char**. We would not want to do this! In this case we say that $\mathbf{delta_1}$ is *worthless* and slide **pat** forward by **k** = 1 (which is always sound). This shifts our attention down **string** by 1+**m**. If the rightmost occurrence of **char** in **pat** is to the left of the mismatch, we can slide forward by **k** = $\mathbf{delta_1(char)-m}$ to align the two occurrences of **char**. This shifts our attention down **string** by $\mathbf{delta_1(char)-m+m}$ = $\mathbf{delta_1(char)}$.

However, it is possible that we can do better than this.

Observation 3-b: We know that the next **m** characters of **string** match the final **m** characters of **pat**. Let this substring of **pat** be **subpat**. We also know that this occurrence of **subpat** in **string** is preceded by a character (**char**) which is different from the character preceding the terminal occurrence of **subpat** in **pat**. Roughly speaking, we can generalize the kind of reasoning used above and can slide **pat** down by some amount so that the discovered occurrence of **subpat** in **string** is aligned with the rightmost occurrence of **subpat** in **pat** which is not preceded by the character preceding its terminal occurrence in **pat**. We will call such a reoccurrence of **subpat** in **pat** a "plausible reoccurrence". The reason we said "roughly speaking" above is that we must allow for the rightmost plausible reoccurrence of **subpat** to "fall off" the left end of **pat**. This is made precise later.

Therefore, according to **Observation 3-b**, if we have matched the last **m** characters of **pat** before finding a mismatch, we can move **pat** down by **k** characters, where **k** is based on the position, in **pat**, of the rightmost plausible reoccurrence of the terminal substring of **pat** having **m** characters. After sliding down by **k** we will want to inspect the character of **string** aligned with the last character of **pat**. Thus, we will actually shift our attention down **string** by **k+m** characters. We will call this distance $\mathbf{delta_2}$, and we will define $\mathbf{delta_2}$ as a function of the position, **j**, in **pat**, at which the mismatch occurred. **k** is just the distance between the terminal occurrence of **subpat** and its rightmost plausible reoccurrence and is always greater than or equal to 1. **m** is just **patlen-j**.

In the case where we have matched the final **m** characters of **pat** before failing, we clearly wish to shift our attention down **string** by **1+m** or $\mathbf{delta_1(char)}$ or $\mathbf{delta_2(j)}$, according to whichever allows the largest shift.

From the definition of $\mathbf{delta_2}$ as **k+m** where **k** is always greater than or equal to 1, it is clear that $\mathbf{delta_2}$ is at least as large as **1+m**. Therefore, we can shift our attention down **string** by the maximum of just the two **deltas**. This rule also applies when **m=0** (i.e., when we have not yet matched any characters of **pat**) because in that case **j**=**patlen** and **delta**$_2$**(j)**$\geq$1.

An Example

In the following example we use an "↑" under **string** to indicate the current **char**. When this "pointer" is pushed to the right, imagine that it drags the right end of **pat** with it (i.e., imagine **pat** has a hook on its right end). When the pointer is moved to the left, keep **pat** fixed with respect to **string**.

```
pat:          AT-THAT
string:   ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                    ↑
```

Since 'F' is known not to occur in **pat**, we can appeal to **Observation 1** and move the pointer (and thus **pat**) down by 7:

```
pat:                 AT-THAT
string:   ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                           ↑
```

Appealing to **Observation 2**, we can move the pointer down 4 to align the two hyphens:

```
pat:                      AT-THAT
string:    ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                ↑
```

Now **char** matches its opposite in **pat**. Therefore we step left by one:

```
pat:                     AT-THAT
string:   ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                              ↑
```

Appealing to **Observation 3-a**, we can move the pointer to the right by 7 positions because 'L' does not occur in **pat**[1]. Note that this only moves **pat** to the right by 6.

```
pat:                           AT-THAT
string:   ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                     ↑
```

Again **char** matches the last character of **pat**. Stepping to the left we see that the previous character in **string** also matches its opposite in **pat**. Stepping to the left a second time produces:

```
pat:                           AT-THAT
string:   ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                   ↑
```

Noting that we have a mismatch, we appeal to **Observation 3-b**. The **delta$_2$** move is best since it allows us to push the pointer to the right by 7, so as to align the discovered substring "AT" with the beginning of **pat**[1].

```
pat:                                AT-THAT
string:   ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                          ↑
```

This time we discover that each character of **pat** matches the corresponding character in **string** so we have found the pattern. Note that we made only 14 references to **string**. Seven of these were required to confirm the final

---

[1]Note that **delta$_2$** would allow us to move the pointer to the right only 4 positions, in order to align the discovered substring "T" in **string** with its second from last occurrence at the beginning of the word "THAT" in **pat**.

---

[1]The **delta$_1$** move only allows the pointer to be pushed to the right by 4 to align the hyphens.

match. The other seven allowed us to move past the first 22 characters of **string.**

The Algorithm

We will now specify the algorithm. The notation **pat(j)** refers to the j^th^ character in **pat** (counting from 1 on the left).

We assume the existence of two tables, **$delta_1$** and **$delta_2$**. The first has as many entries as there are characters in the alphabet. The entry for some character **char** will be denoted by **$delta_1$(char)**. The second table has as many entries as there are character positions in the pattern. The j^{th} entry will be denoted by **$delta_2$(j)**. Both tables contain non-negative integers.

The tables are initialized by preprocessing **pat** and their entries correspond to the values **$delta_1$** and **$delta_2$** referred to earlier. We will specify their precise contents after it is clear how they are to be used.

Our search algorithm may be specified as follows:

```
          stringlen ← length of string.
          i ← patlen.
top:      if i > stringlen then return false.
          j ← patlen.
loop:     if j=0 then return i+1.
          if string(i) = pat(j)
               then
               j ← j-1.
               i ← i-1.
               goto loop.
               close;
          i ← i + max($delta_1$(string(i)),$delta_2$(j)).
          goto top.
```

If the above algorithm returns false then **pat** does not occur in **string.** If the algorithm returns a number, then it is the position of the left end of the first occurrence of **pat** in **string**.

The **$delta_1$** table has an entry for each character, **char**, in the alphabet. The definition of **$delta_1$** is:

$delta_1$(char) = if **char** does not occur in **pat**, then **patlen**; else **patlen-j**, where **j** is the maximum integer such that **pat(j)** = **char**.

The **$delta_2$** table has one entry for each of the integers from 1 to **patlen**. Roughly speaking, **$delta_2$(j)** is (1) the distance we can slide **pat** down so as to align the discovered occurrence (in **string**) of the last **patlen-j** characters of **pat** with its rightmost plausible reoccurrence, plus (2) the additional distance we must slide the "pointer" down so as to restart the process at the right end of **pat**. To define **$delta_2$** precisely we must define the rightmost plausible reoccurrence of a terminal substring of **pat**. To this end let us make the following conventions: Let $ be a character that does not occur in **pat** and let us say that if **i** is less than 1 then **pat(i)** is $. Let us also say that two sequences of characters, [c_1 ... c_n] and [d_1 ... d_n], *unify* if for all i from 1 to n either $c_i = d_i$ or c_i = $ or d_i = $.

Finally, we define the position of the rightmost plausible reoccurrence of the terminal substring which starts at position j+1, rpr(j), for **j** from 1 to **patlen**, to be the greatest **k** less than or equal to **patlen** such that [pat(j+1) ... pat(patlen)] and [pat(k) ... pat(k+patlen-j-1)] unify and either **k** $\leq$ 1 or **pat(k-1)** $\neq$ **pat(j)**[1]. (That is, the position of the rightmost plausible reoccurrence of the substring, **subpat**, which starts at j+1, is the rightmost place where **subpat** occurs in **pat** and is not preceded by the character, **pat(j)**, which precedes its terminal occurrence -- with suitable allowances for either the reoccurrence or the preceding character to fall beyond the left end of **pat**. Note that **rpr(j)** may be negative because of these allowances.)

Thus, the distance we must slide **pat** to align the discovered substring which starts at j+1 with its rightmost plausible reoccurrence is **j+1-rpr(j)**. The distance we must move to get back to the end of **pat** is just **patlen-j**. **delta_2(j)** is just the sum of these two. Thus we define **delta_2** as follows:

$$\text{delta}_2(j) = \quad \text{patlen}+1-\text{rpr}(j).$$

To make this definition clear, consider the following two examples:

j:	1	2	3	4	5	6	7	8	9
pat:	A	B	C	X	X	X	A	B	C
delta_2(j):	14	13	12	11	10	9	11	10	1

j:	1	2	3	4	5	6	7	8	9
pat:	A	B	Y	X	C	D	E	Y	X
delta_2(j):	17	16	15	14	13	12	7	10	1

Implementation Considerations

The most frequently executed part of the algorithm is the code that embodies **Observations 1** and **2**. The following version of our algorithm is equivalent to the original version provided that **delta_0** is a table containing

---

[1]Note that when j=**patlen**, the two sequences [pat(patlen+1) ... pat(patlen)] and [pat(k) ... pat(k-1)] are empty and therefore unify. Thus, **rpr(patlen)** is simply the greatest **k** less than or equal to **patlen** such that $k \leq 1$ or $\text{pat}(k-1) \neq \text{pat}(\text{patlen})$.

the same entries as **$delta_1$** except that **$delta_0$(pat(patlen))** is set to an integer, **large**, which is greater than **stringlen+patlen** (while **$delta_1$(pat(patlen))** is always 0).

```
         stringlen ← length of string.
         i ← patlen.
fast:    i ← i + delta₀(string(i)).
         if i ≤ stringlen then goto fast.
undo:    if i ≤ large then return false.
         i ← (i-large)-1.
         j ← patlen-1.
slow:    if j=0 then return i+1.
         if string(i) = pat(j)
             then
             j ← j-1.
             i ← i-1.
             goto slow.
             close;
         i ← i + max(delta₁(string(i)),delta₂(j)).
         goto fast.
```

Of course, we do not actually have two versions of **$delta_1$**. Instead we use only **$delta_0$** and in place of **$delta_1$** in the **max** expression we merely use the **$delta_0$** entry unless it is **large** (in which case we use 0).

Note that the **fast** loop just scans down **string** effectively looking for the last character, **pat(patlen)**, in **pat**, skipping according to **$delta_1$**. (**$delta_2$** can be ignored in this case since no terminal substring has yet been matched, i.e., **$delta_2$(patlen)** is always less than or equal to the corresponding **$delta_1$**.) Control leaves this loop only when **i** exceeds **stringlen**. The test at **undo** decides whether this situation arose because all of **string** has been scanned or because **pat(patlen)** was hit (which caused **i** to be incremented by **large**). If the first case obtains, **pat** does not occur in **string** and the algorithm returns false. If the second case obtains then **i** is restored (by subtracting **large**) and we enter the **slow** loop which backs up checking for matches. When a mismatch is found we skip ahead by the maximum of the original **$delta_1$** and **$delta_2$** and reenter the **fast** loop. We estimate that 80 percent of the time spent in searching is spent in the **fast** loop.

The **fast** loop can be coded in 4 machine instructions:

```
fast:    char ← string(i).
         i ← i + delta_0(char).
         skip the next instruction if i > stringlen.
         goto fast.
undo:    ...
```

We have implemented this algorithm in PDP-10 assembly language. In our implementation we have reduced the number of instructions in the **fast** loop to 3 by translating **i** down by **stringlen**; we can then test i against 0 and conditionally jump to **fast** in one instruction.

On a byte addressable machine it is easy to implement "**char ← string(i)**" and "**i ← i + $delta_0$(char)**" in one instruction each. Since our implementation was in PDP-10 assembly language we had to employ byte pointers to access characters in **string**. The PDP-10 instruction set provides an instruction for incrementing a byte pointer by one but not by other amounts. Our code therefore employs an array of 200 indexing byte pointers which we use to access characters in **string** in one indexed instruction (after computing the index) at the cost of a small (5 instruction) overhead every 200 characters. It should be noted that this trick only makes up for the lack of direct byte addressing; one can expect our algorithm to run somewhat faster on a byte addressable machine.

Empirical Evidence

We have exhaustively tested the above PDP-10 implementation on random test data. To gather the test patterns we wrote a program which randomly selects a substring of a given length from a source string. We used this program to select 300 patterns of length **patlen**, for each **patlen** from 1 to 14. We then used our algorithm to search for each of the test patterns in its source string, starting each search in a random position somewhere in the first half of the source string. All of the characters for both the patterns and the strings were in primary memory (rather than a secondary storage medium such as a disk).

We measured the cost of each search in two ways: the number of references made to **string**, and the total number of machine instructions that actually got executed (ignoring the preprocessing to set up the two tables).

By dividing the number of references to **string** by the number of characters, **i-1**, passed before the pattern was found (or **string** was exhausted) we obtained the number of references to **string** per character passed. This measure is independent of the particular implementation of the algorithm. By dividing the number of instructions executed by **i-1** we obtained the average number of instructions spent on each character passed. This measure depends upon the implementation, but we feel that it is meaningful since the implementation is a straightforward encoding of the algorithm as described in the last section.

We then averaged these measures across all 300 samples for each pattern length.

Because the performance of the algorithm depends upon the statistical properties of **pat** and **string** (and hence upon the properties of the source string from which the test patterns were obtained) we performed this experiment for three different kinds of source strings, each of length 10000. The first source string consisted of a random sequence of 0's and 1's. The second source string was a piece of English text obtained from an online manual. The third source string was a random sequence of characters from a 100 character alphabet.

In Figure 1, the average number of references to **string** per character in **string** passed is plotted against the pattern length for each of three source strings.

Note that the number of references to **string** per character passed is less than 1. For example, for an English pattern of length 5 the algorithm typically inspects 0.24 characters for every character passed. That is, for every reference to **string** the algorithm passes about 4 characters, or, equivalently, the algorithm inspects only about a quarter of the characters it passes when searching for a pattern of length 5 in an English text string. Furthermore, the number of references per character drops as the patterns get longer. This evidence supports the conclusion that the algorithm is "sublinear" in the number of references to **string**.

For comparison, it should be noted that the Knuth, Morris, and Pratt algorithm references **string** precisely 1 time per character passed. The simple search algorithm references **string** about 1.1 times per character passed (determined empirically with the English sample above).

In Figure 2 the average number of instructions executed per character passed is plotted against the pattern length.

The most obvious feature to note is that the search speeds up as the patterns get longer. That is, the total number of instructions executed in order to pass over a character decreases as the length of the pattern increases.

Figure 2 also exhibits a second interesting feature of our implementation of the algorithm: for sufficiently large alphabets and sufficiently long patterns the algorithm executes fewer than 1 instruction per character passed. For example, in the English sample, less than 1 instruction per character is executed for patterns of length 5 or more. Thus, this implementation is "sublinear" in the sense that it executes fewer than **i+patlen** instructions before finding the pattern at **i**.

This means that no algorithm which references each character it passes could *possibly* be faster than ours in these cases (assuming it takes at least one instruction to reference each character).

The best alternative algorithm for finding a single substring is that of Knuth, Morris, and Pratt. If that algorithm is implemented in the extraordinarily efficient way described in [4] (pg. 11-12) and [2] (Item 179)[1] then the cost of looking at a character can be expected to be at least 3-p instructions, where p is the probability that a character just fetched from **string** is equal to a given character of **pat**. Hence, a horizontal line at 3-p instructions/character represents the best (and, practically, the worst) the Knuth, Morris, and Pratt algorithm can achieve.

----
[1]This implementation automatically compiles **pat** into a machine code program which implicitly has the skip table built in and which is executed to perform the search itself. In [2] they compile code which uses the PDP-10 capability of fetching a character and incrementing a byte address in one instruction. This compiled code executes at least 2 or 3 instructions per character fetched from **string**, depending on the outcome of a comparison of the character to one from **pat**.

The simple string searching algorithm (when coded with a 3-instruction fast loop[1]) executes about 3.3 instructions per character (determined empirically on the English sample above).

As noted above, the preprocessing time for our algorithm (and for Knuth, Morris, and Pratt) has been ignored. The cost of this preprocessing can be made linear in **patlen** (this is discussed further in the next section) and is trivial compared to a reasonably long search. We made no attempt to code this preprocessing efficiently. However the average cost (in our implementation) ranges from 160 instructions (for strings of length 1) to about 500 instructions (for strings of length 14). It should be explained that our code uses a block transfer instruction to clear the 128-word $\mathbf{delta_1}$ table at the beginning of the preprocessing, and we have counted this single instruction as though it were 128 instructions. This accounts for the unexpectedly large instruction count for preprocessing a one character pattern.

Theoretical Analysis

The preprocessing for $\mathbf{delta_1}$ requires an array the size of the alphabet. Our implementation first initializes all entries of this array to **patlen** and then sets up $\mathbf{delta_1}$ in a linear scan through the pattern. Thus, our preprocessing for $\mathbf{delta_1}$ is linear in **patlen** plus the size of the alphabet.

----

[1]This loop avoids checking whether **string** is exhausted by assuming that the first character of **pat** occurs at the end of **string**. This can be arranged ahead of time. The loop actually uses the same three instruction codes used by the above referenced implementation of the Knuth, Morris, and Pratt algorithm.

At a slight loss of efficiency in the search speed one could eliminate the initialization of the **$delta_1$** array by storing with each entry a key indicating the number of times the algorithm has previously been called. This approach still requires initializing the array the first time the algorithm is used.

To implement our algorithm for extremely large alphabets, one might implement the **$delta_1$** table as a hash array. In the worst case, then, accessing **$delta_1$** during the search itself could require order **patlen** instructions, significantly impairing the speed of the algorithm. As noted below, Knuth has shown that the execution time of the algorithm is linear in **i+patlen**, even if **$delta_1$** is ignored. However, this would drastically degrade the performance of the algorithm on the average.

In [5] Knuth exhibits an algorithm for setting up **$delta_2$** in time linear in **patlen.**

From the preceding empirical evidence the reader can conclude that the algorithm is quite good in the average case. However, the question of its behavior in the worst case is non-trivial. Knuth has recently shed some light on this question. In [5] he proves that the execution of the algorithm (after preprocessing) is linear in **i+patlen**, assuming the availability of array space linear in **patlen** plus the size of the alphabet. In particular, he shows that in order to discover that **pat** does not occur in the first **i** characters of **string**, at most 6*i characters from **string** are matched with characters in **pat**. He goes on to say that the constant 6 is probably much too large, and invites the reader to improve the theorem. His proof reveals that the linearity of the algorithm is entirely due to **$delta_2$**.

We now will analyze the average behavior of the algorithm by presenting a probabilistic model of its performance. As will become clear, the results of this analysis will support the empirical conclusions that the algorithm is usually "sublinear" both in the number of references to **string** and the number of instructions executed (for our implementation).

The analysis below is based on the following simplifying assumption: Each character of **pat** and **string** is an independent random variable. The probability that a character from **pat** or **string** is equal to a given character of the alphabet is p.

Imagine that we have just moved **pat** down **string** to a new position and that this position does not yield a match. We want to know the expected value of the ratio between the cost of discovering the mismatch and the distance we get to slide **pat** down upon finding the mismatch. If we define the cost to be the total number of references made to **string** before discovering the mismatch we can obtain the expected value of the average number of references to string per character passed. If we define the cost to be the total number of machine instructions executed in discovering the mismatch we can obtain the expected value of the number of instructions executed per character passed.

In the following we will say that "only the last **m** characters of **pat** match" to mean "the last **m** characters of **pat** match the corresponding **m** characters in **string** but the m+1st character from the right end of **pat** fails to match the corresponding character in **string**".

The expected value of the ratio of cost to characters passed is given by:

$$\frac{\sum_{m=0}^{patlen-1} cost(m) * prob(m)}{\sum_{m=0}^{patlen-1} prob(m) * (\sum_{k=1}^{patlen} skip(m,k)*k)}$$

where **cost(m)** is the cost associated with discovering that only the last **m** characters of **pat** match; **prob(m)** is the probability that only the last **m** characters of **pat** match; and **skip(m,k)** is the probability that, supposing only the last **m** characters of **pat** match, we will get to slide **pat** down by **k**.

Under our assumptions, the probability that only the last **m** characters of **pat** match is:

$$prob(m) = p^{m}(1-p)/(1-p^{patlen}).$$

(The denominator is due to the assumption that a mismatch exists.)

The probability that we will get to slide **pat** down by **k** is determined by analyzing how **i** is incremented. However note that even though we increment **i** by the maximum, **max**, of the two **deltas** this will actually only slide **pat** down by **max-m**, since the increment of **i** also includes the **m** necessary to shift our attention back to the end of **pat**. Thus, when we analyze the contributions of the two **deltas** we will speak of the amount by which they allow us to slide **pat** down, rather than the amount by which we increment **i**. Finally, recall that if the mismatched character, **char,** occurs in the already matched final **m** characters of **pat**, then $\mathbf{delta_1}$ is worthless and we will always slide by $\mathbf{delta_2}$.

The probability that $\mathbf{delta_1}$ is worthless is just $(1-(1-p)^{m})$. Let us call this $\mathbf{probdelta_1worthless(m)}$.

The conditions under which **$\mathbf{delta}_1$** will naturally let us slide forward by **k** can be broken down into four cases as follows: (1) **$\mathbf{delta}_1$** will let us slide down by 1 if **char** is the m+2nd character from the righthand end of **pat** (or else there are no more characters in **pat**) and **char** does not occur to the right of that position (which has probability $(1\text{-}p)^{m}$*(if **m+1=patlen** then 1 else p)). (2) **$\mathbf{delta}_1$** allows us to slide down **k**, where $1 < k <$ **patlen-m**, provided the rightmost occurrence of **char** in **pat** is **m+k** characters from the right end of **pat** (which has probability $p*(1\text{-}p)^{k+m-1}$). (3) When **patlen-m** > 1, **$\mathbf{delta}_1$** allows us to slide past **patlen-m** characters if **char** does not occur in **pat** at all (which has probability $(1\text{-}p)^{patlen-1}$ given that we know **char** is not the **m+1st** character from the right end of **pat**). Finally, (4) **$\mathbf{delta}_1$** never allows a slide longer than **patlen-m** (since the maximum value of **$\mathbf{delta}_1$** is **patlen**).

Thus we can define the probability, **$\mathbf{probdelta}_1$(m,k)**, that, when only the last **m** characters of **pat** match, **$\mathbf{delta}_1$** will allow us to move down by **k** as follows:

```
probdelta_1(m,k) =  if k=1
                          then
                          $(1-p)^{m}$*(if m+1=patlen then 1 else p);
                    elseif 1<k<patlen-m then $p*(1-p)^{k+m-1}$.
                    elseif k=patlen-m then $(1-p)^{patlen-1}$;
                    else (i.e. k>patlen-m) 0.
```

(It should be noted that we will not put these formulas into closed form, but will simply evaluate them to verify the validity of our empirical evidence.)

We will now perform a similar analysis for **delta$_2$**. **delta$_2$** lets us slide down by **k** if (1) doing so sets up an alignment of the discovered occurrence of the last **m** characters of **pat** in **string** with a plausible reoccurrence of those **m** characters elsewhere in **pat**, and (2) no smaller move will set up such an alignment. The probability, **probpr(m,k)**, that the terminal substring of **pat** of length **m** has a plausible reoccurrence **k** characters to the left of its first character is:

```
probpr[m,k] = if m+k < patlen
                  then $(1-p)*p^{m}$
                  else $p^{patlen-k}$
```

Of course, **k** is just the distance **delta$_2$** would let us slide, provided there is no earlier reoccurrence. We can therefore define the probability, **probdelta$_2$(m,k)**, that, when only the last **m** characters of **pat** match, **delta$_2$** will allow us to move down by **k** recursively as follows:

$$\mathrm{probdelta}_2(m,k) = \mathrm{probpr}(m,k)(1-\sum_{n=1}^{k-1} \mathrm{probdelta}_2(m,n)),$$

We will slide down by the maximum allowed by the two **deltas** (taking adequate account of the possibility that **delta$_1$** is worthless). If the values of the **deltas** were independent, the probability that we would actually slide down by **k** would just be the sum of the products of the probabilities that one of the **deltas** allows a move of **k** while the other allows a move of less than or equal to **k**.

However, the two moves are not entirely independent. In particular, consider the possibility that **delta$_1$** is worthless. Then the **char** just fetched occurs in the last **m** characters of **pat** and does not match the

m+1st. But if **delta$_2$** gives a slide of 1 it means that sliding these **m** characters to the left by 1 produces a match. This implies that all of the last **m** characters of **pat** are equal to the character **m**+1 from the right. But this character is known not to be **char**. Thus, **char** cannot occur in the last **m** characters of **pat**, violating the hypothesis that **delta$_1$** was worthless. Therefore, if **delta$_1$** is worthless, the probability that **delta$_2$** specifies a skip of 1 is 0 and the probability that it specifies one of the larger skips is correspondingly increased.

This interaction between the two **deltas** is also felt (to a lesser extent) for the next **m** possible **delta$_2$**'s but we will ignore these (and in so doing accept that our analysis may predict slightly worse results than might be expected since we will be allowing some short **delta$_2$** moves when longer ones would actually occur).

The probability that **delta$_2$** will allow us to slide down by **k** when only the last **m** characters of **pat** match, assuming that **delta$_1$** is worthless, is:

```
probdelta$_2$'(m,k) = if k=1
                          then 0
                          else
                          probpr(m,k)(1-$\sum_{n=2}^{k-1}$ probdelta$_2$'(m,n)).
```

Finally, we can define **skip(m,k)**, the probability that we will slide down by **k** if only the last **m** characters of **pat** match:

```
skip(m,k) =   if k=1
                  then probdelta$_1$(m,1)*probdelta$_2$(m,1)
                  else
                  probdelta$_1$worthless(m)*probdelta$_2$'(m,k)
                              +
                  k-1
                  $\Sigma$ probdelta$_1$(m,k)*probdelta$_2$(m,n)
                  n=1
                              +
                  k-1
                  $\Sigma$ probdelta$_1$(m,n)*probdelta$_2$(m,k)
                  n=1
                              +
                  probdelta$_1$(m,k)*probdelta$_2$(m,k).
```

Now let us consider the two alternative **cost** functions. In order to analyze the number of references to **string** per character passed over, **cost(m)** should just be **m**+1, the number of references necessary to confirm that only the last **m** characters of **pat** match.

In order to analyze the number of instructions executed per character passed over, **cost(m)** should be the total number of instructions executed in discovering that only the last **m** characters of **pat** match. By inspection of our PDP-10 code:

```
cost(m) =     if m=0 then 3 else 12+6m.
```

We have computed the expected value of the ratio of cost per character skipped using the above formulas (and both definitions of **cost**). We did so for pattern lengths running from 1 to 14 (as in our empirical evidence) and for the values of p appropriate for the three source strings used: For a random binary string p is 0.5, for an arbitrary English string it is (approximately) 0.09, and for a random string over a 100 character alphabet it is 0.01. The value of p for English was determined using a standard frequency count for the alphabetic characters [3] and empirically determining the frequency of space, carriage return, and line feed to be 0.23, 0.03, and 0.03 respectively[1].

In Figure 3 we have plotted the theoretical ratio of references to **string** per character passed over against the pattern length.

The most important fact to observe in Figure 3 is that the algorithm *can be expected* to make fewer than **i+patlen** references to **string** before finding the pattern at location **i**. For example, for English text strings of length 5 or greater, the algorithm may be expected to make less than (i+5)/4 references to **string**. The comparable figure for the Knuth, Morris, and Pratt algorithm is of course precisely **i**. The figure for the intuitive search algorithm is always greater than or equal to **i**.

The reason the number of references per character passed decreases more slowly as **patlen** increases is that for longer patterns the probability is higher that the character just fetched occurs somewhere in the pattern, therefore shortening the distance the pattern can be moved forward.

In Figure 4 we have plotted the theoretical ratio of the number of instructions executed per character passed versus the pattern length.

Again we find that our implementation of the algorithm *can be expected*

----

[1]We have determined empirically that the algorithm's performance on truly random strings where p=0.09 is virtually identical to its performance on English strings. In particular, the reference count and instruction count curves generated by such random strings are almost coincidental with the English curves in Figures 1 and 2.

(for sufficiently large alphabets) to execute fewer than **i+patlen** instructions before finding the pattern at location **i**. That is, our implementation is usually "sublinear" even in the number of instructions executed. The comparable figure for the Knuth, Morris, and Pratt algorithm is at best (3-p)*(i+patlen-1)[1]. For the simple search algorithm the expected value of the number of instructions executed per character passed is (approximately) 3.28 (for p=0.09).

It is difficult to fully appreciate the role played by **$delta_2$**. For example, if the alphabet is large and patterns are short then computing and trying to use **$delta_2$** probably does not pay off much (because the chances are high that a given character in **string** does not occur anywhere in **pat** and one will almost always stay in the **fast** loop ignoring **$delta_2$**)[2]. Conversely, **$delta_2$** becomes very important when the alphabet is small and the patterns are long (for now execution will frequently leave the **fast** loop, **$delta_1$** will in general be small because many of the characters in the alphabet will occur in **pat**, and only the terminal substring observations could cause large shifts). Despite the fact that it is difficult to appreciate the role of **$delta_2$** it should be noted that the linearity result for the worst case behavior of the algorithm is due entirely to the presence of **$delta_2$**.

----

[1]Although the Knuth, Morris, and Pratt algorithm will fetch each of the first **i+patlen-1** characters of **string** precisely once, sometimes a character is involved in several tests against characters in **pat**. The number of such tests (each involving 3 instructions) is bounded by **$\log_\Phi$(patlen)**, where Φ is the golden ratio.

[2]However, if the algorithm is implemented without **$delta_2$** recall that, in exiting the **slow** loop, one must now take the **max** of **$delta_1$** and **patlen-j+1** to allow for the possibility that **$delta_1$** is worthless.

If we compare the empirical evidence (Figures 1 and 2) with the theoretical evidence (Figures 3 and 4, respectively) we will note that the model is completely accurate for English and the 100 character alphabet. The model predicts much better behavior than we actually experience in the binary case. Our only explanation is that since **$\mathbf{delta}_2$** predominates in the binary alphabet, and since it sets up alignments of the pattern and the string, the algorithm backs up over longer terminal substrings of the pattern before finding mismatches. Our analysis ignores this phenomenon.

However, in summary, the theoretical analysis supports the conclusion that on the average the algorithm is sublinear in the number of references to **string**, and, for sufficiently large alphabets and patterns, sublinear in the number of instructions executed (in our implementation).

Caveat Programmer

It should be observed that the preceding analysis has assumed that **string** is entirely in primary memory and that we can obtain the ith character in it in one instruction after computing its byte address. However, if **string** is actually on secondary storage then the characters in it must be read in[1]. This transfer will entail some time delay equivalent to the execution of,

---

[1]We have implemented a version of our algorithm for searching through disk files. It is available as the subroutine FFILEPOS in the latest release of INTERLISP-10. This function uses the TENEX page mapping capability to identify one file page at a time with a buffer area in virtual memory. In addition to being faster than reading the page by conventional methods, this means the operating system's memory management takes care of references to pages which happen to still be in memory, etc. The algorithm is as much as 50 times faster than the standard INTERLISP-10 FILEPOS function (depending on the length of the pattern).

say, w instructions per character brought in, and (because of the nature computer i/o) all of the first **i+patlen-1** characters will eventually be brought in whether *we* actually reference all of them or not. (A representative figure for w for paged transfers from a fast disk is 5 instructions/character.) Thus, there may be a hidden cost of w instructions per character passed over.

Now according to the statistics presented above one might expect our algorithm to be approximately 3 times faster than the Knuth, Morris, and Pratt algorithm (for, say, English strings of length 6), since that algorithm executes about 3 instructions to our 1. However, if the cpu is idle for the w instructions necessary to read each character the actual ratios are closer to w+3 instructions to w+1 instructions. (Thus, for paged disk transfers our algorithm can only be expected to be roughly 4/3 faster (i.e. 5+3 instructions to 5+1 instructions) if we assume that we are idle during i/o.) Thus, for large values of w the difference between the various algorithms diminishes if the cpu is idle during i/o.

Of course, in general, programmers (or operating systems) try to avoid the situation in which the cpu is idle while awaiting an i/o transfer by overlapping i/o with some other computation. In this situation, the chances are that our algorithm will be i/o bound (we will search a page faster than it can be brought in), and indeed, so will that of Knuth, Morris, and Pratt if w > 3. Our algorithm will require that fewer cpu cycles be devoted to the search itself, so that if there are other jobs to perform, there is still an overall advantage in using the algorithm.

There are several situations in which it may not be advisable to use our algorithm. If the expected penetration, **i**, at which the pattern is found is small, the preprocessing time is significant and one might therefore consider using the obvious intuitive algorithm.

As previously noted, our algorithm can be most efficiently implemented on a byte-addressable machine. On a machine that does not allow byte addresses to be incremented and decremented directly, two possible sources of inefficiency must be addressed: The algorithm typically skips through **string** in steps larger than 1, and the algorithm may back up through **string**. Unless these processes are coded efficiently it is probably not worthwhile to use our algorithm.

Furthermore, it should be noted that, because the algorithm can back up through **string**, it is possible to cross a page boundary more than once. We have not found this to be a serious source of inefficiency. However, it does require a certain amount of code to handle the necessary buffering (if paged i/o is being handled directly as in our FFILEPOS). One beauty of the Knuth, Morris, and Pratt algorithm is that it avoids this problem altogether.

A final situation in which it is unadvisable to use our algorithm is if the string matching problem to be solved is actually more complicated than merely finding the first occurrence of a single substring. For example, if the problem is to find the first of several possible substrings, or to identify a location in **string** defined by a regular expression it is much more advantageous to use an algorithm such as that of Aho and Corasick [1].

It may of course be possible to design an algorithm that searches for multiple patterns or instances of regular expressions using the idea of starting the match at the right end of the pattern. However, we have not designed such an algorithm.

Historical Remarks

Our earliest formulation of the algorithm involved only **$delta_1$** and implemented **Observations 1, 2,** and **3-a.** We were aware that we could do something along the lines of **$delta_2$** and **Observation 3-b,** but did not precisely formulate it. Instead, in April, 1974, we coded the **$delta_1$** version of the algorithm in INTERLISP, merely to test its speed. We considered coding the algorithm in PDP-10 assembly language but abandoned the idea as impractical because of the cost of incrementing byte pointers by arbitrary amounts.

We have since learned that R. W. Gosper, of Stanford University, simultaneously and independently discovered the **$delta_1$** version of the algorithm (private communication).

In April, 1975, we started thinking about the implementation again and discovered a way to increment byte pointers by indexing through a table. We then formulated a version of **$delta_2$** and coded the algorithm more or less as it is presented here. This original definition of **$delta_2$** differed from the current one in the following respect: If only the last **m** characters of **pat** (call this substring **subpat**) were matched, **$delta_2$** specified a slide to the second from the rightmost occurrence of **subpat** in **pat** (allowing this occurrence to "fall off" the left end of **pat**) but without any special consideration of the character preceding this occurrence.

The average behavior of that version of the algorithm was virtually indistinguishable from that presented in this paper for large alphabets, but was somewhat worse for small alphabets. However, its worst case behavior was quadratic (i.e., required on the order of **i*patlen** comparisons). For example, consider searching for a pattern of the form $CA(BA)^r$ in a string of the form $((XX)^r(AA)(BA)^r)^*$ (e.g. **r** = 2, **pat** = "CABABA", and **string** = "XXXXAABABAXXXXAABABA..."). The original definition of $\mathbf{delta_2}$ allowed only a slide of 2 if the last "BA" of **pat** were matched before the next 'A' fails to match. Of course, in this situation this only sets up another mismatch at the same character in **string**, but the algorithm had to reinspect the previously inspected characters to discover it. The total number of references to **string** in passing **i** characters in this situation was $(r+1)*(r+2)*i/(4r+2)$, where **r** = $(patlen-2)/2$. Thus, the number of references was on the order of **i*patlen**.

However, on the average the algorithm was blindingly fast. To our surprise, it was several times faster than the string searching algorithm in the Tenex TECO text editor. This algorithm is reputed to be quite an efficient implementation of the simple search algorithm because it searches for the first character of **pat** one full-word at a time (rather than one byte at a time).

In the summer of 1975, we wrote a brief paper on the algorithm and distributed it on request.

In December, 1975, Ben Kuipers, of the M.I.T. Artificial Intelligence Laboratory, read the paper and brought to our attention the improvement to $\mathbf{delta_2}$ concerning the character preceding the terminal substring and its reoccurrence (private communication). Almost simultaneously, Donald Knuth, of Stanford University, suggested the same improvement and observed that the improved algorithm could certainly make no more than order **(i+patlen)*log(patlen)** references to **string** (private communication).

We mentioned this improvement in the next revision of the paper and suggested an additional improvement, namely the replacement of both $\mathbf{delta}_1$ and $\mathbf{delta}_2$ by a single two dimensional table. Given the mismatched **char** from **string** and the position **j** in **pat** at which the mismatch occurred, this table indicated the distance to the last occurrence (if any) of the substring **[char,pat(j+1), ... pat(patlen)]** in **pat**. The revised paper concluded with the question of whether this improvement or a similar one produced an algorithm which was at worst linear and on the average "sublinear".

In January, 1976, Knuth, [5], proved that the simpler improvement in fact produces linear behavior, even in the worst case. We therefore revised the paper again and gave $\mathbf{delta}_2$ its current definition.

In April, 1976, R. W. Floyd, of Stanford University, discovered a serious statistical fallacy in the first version of our formula giving the expected value of the ratio of cost to characters passed. He provided us (private communication) with the current version of this formula.

Thomas Standish, of the University of California at Irvine, has suggested (private communication) that the implementation of the algorithm can be improved by fetching larger bytes in the **fast** loop (i.e., bytes containing several characters) and using a hash array to encode the extended $\mathbf{delta}_1$ table. Provided the difficulties at the boundaries of the pattern are handled efficiently this could improve the behavior of the algorithm enormously since it exponentially increases the effective size of the alphabet and reduces the frequency of common characters.

Acknowledgments

We would like to thank B. Kuipers, of the M.I.T. Artificial Intelligence Laboratory, for his suggestion concerning **delta$_2$** and D. Knuth, of Stanford University, for his analysis of the improved algorithm. We are grateful to the anonymous reviewer for the CACM who suggested the inclusion of evidence comparing our algorithm with that of Knuth, Morris, and Pratt, and for the warnings contained in **Caveat Programmer**. B. Mont-Reynaud, of the Stanford Research Institute, and L. Guibas of Xerox Palo Alto Research Center, proof read drafts of this paper and suggested several clarifications. We would also like to thank E. Taft and E. Fiala of Xerox Palo Alto Research Center for their advice regarding machine coding the algorithm.

References

[1] Aho, A. V., and Corasick, M. J. Fast pattern matching: an aid to bibliographic search. C. ACM 18 (June, 1975).

[2] Beeler, M., Gosper, R. W., and Schroeppel, R. "HAKMEM,", M.I.T. Artificial Intelligence Laboratory Memo No. 239 (February 29, 1972).

[3] Dewey, G., *Relativ Frequency of English Speech Sounds,* Harvard University Press, Cambridge (1923) pg. 185.

[4] Knuth, D. E., Morris, J. H., and Pratt, V. R. Fast pattern matching in strings. TR CS-74-440, Stanford University, Stanford, California, 1974.

[5] Knuth, D. E., Morris, J. H., and Pratt, V. R. Fast pattern matching in strings. (to appear in the SIAM Journal of Computation).

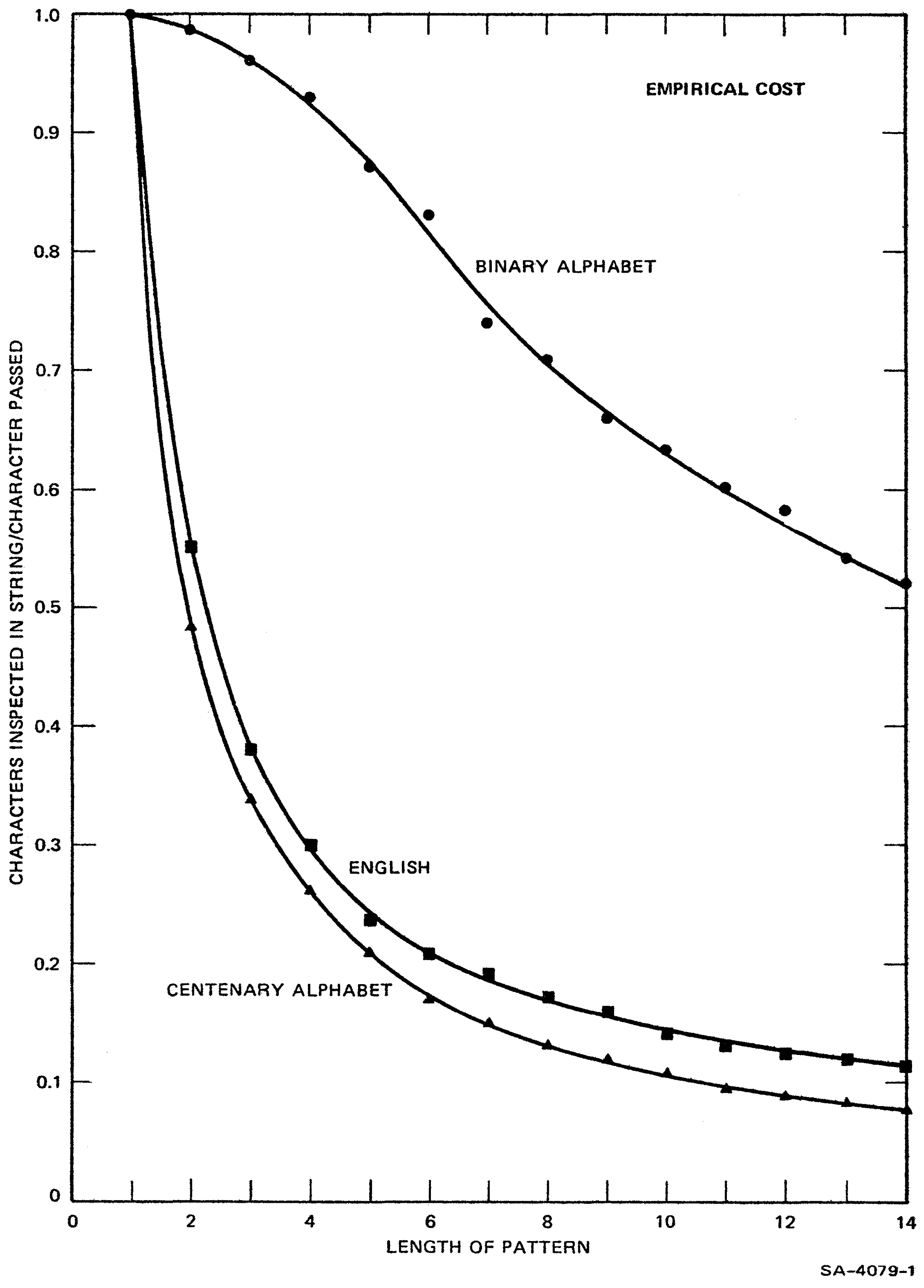
EMPIRICAL COST
BINARY ALPHABET
ENGLISH
CENTENARY ALPHABET
CHARACTERS INSPECTED IN STRING/CHARACTER PASSED
LENGTH OF PATTERN
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
14
SA-4079-1

FIGURE 1

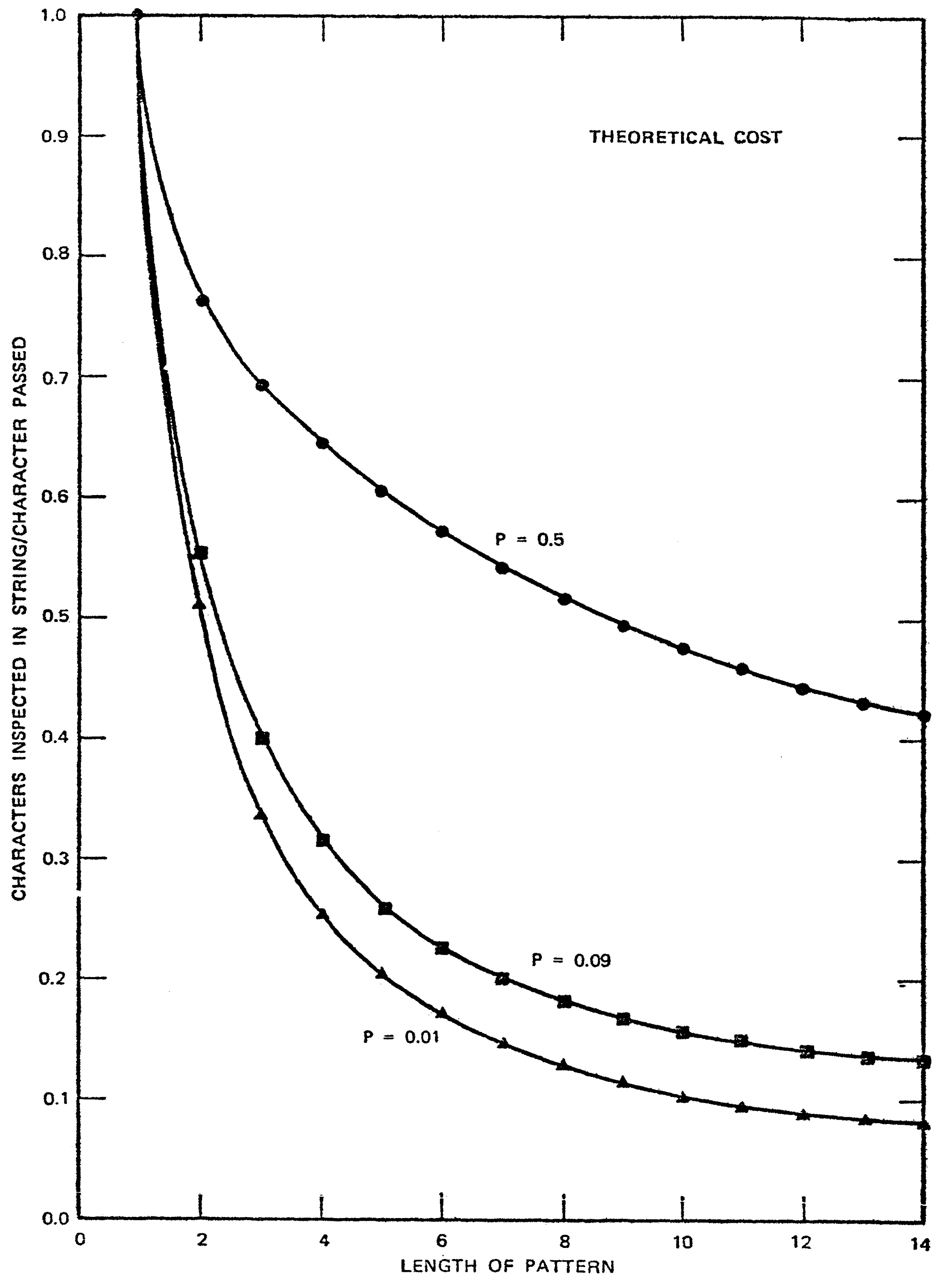
THEORETICAL COST
P = 0.5
P = 0.09
P = 0.01
CHARACTERS INSPECTED IN STRING/CHARACTER PASSED
LENGTH OF PATTERN
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10
12
14
SA-4079-5

FIGURE 3

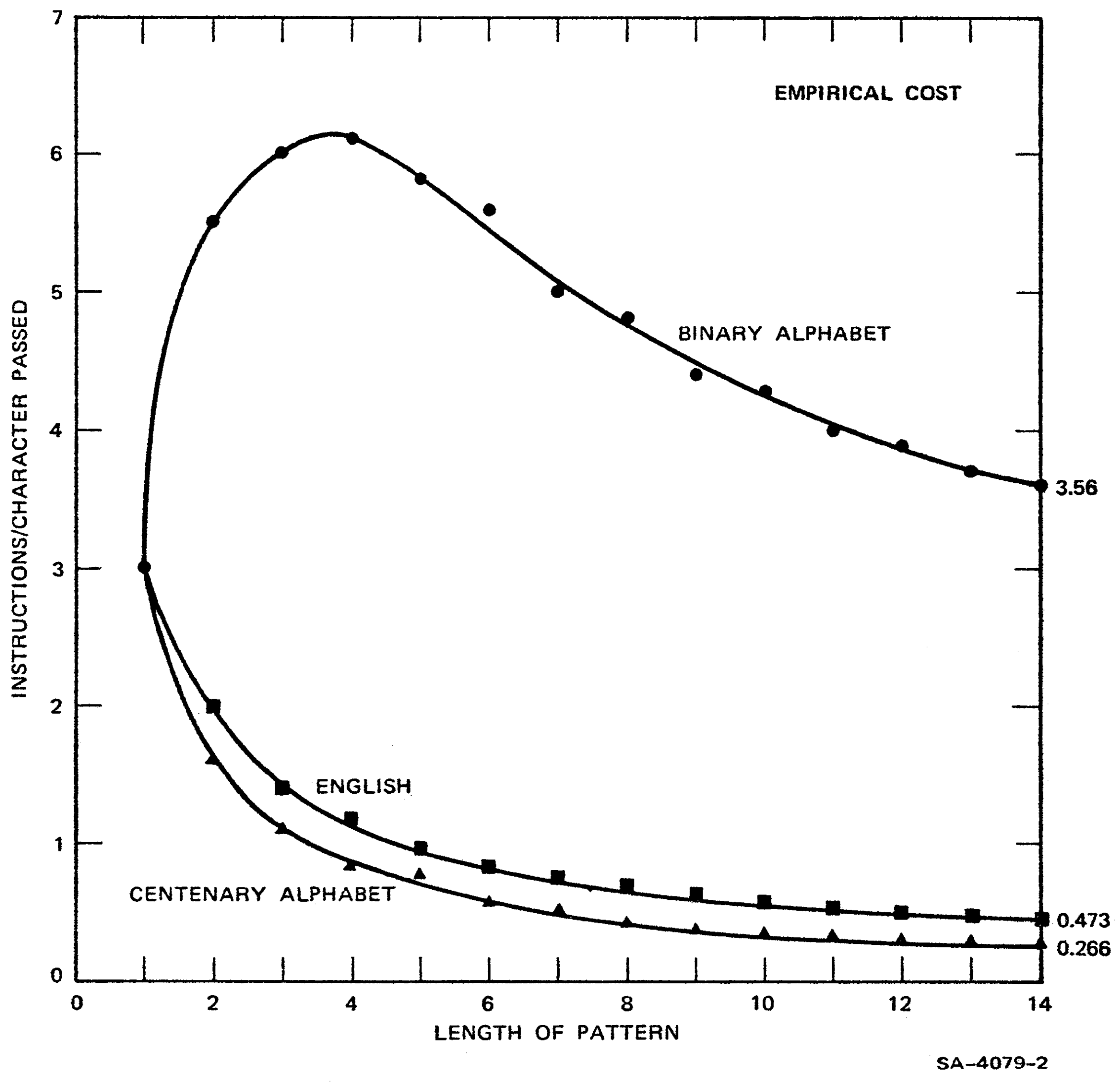
EMPIRICAL COST
BINARY ALPHABET
ENGLISH
CENTENARY ALPHABET
INSTRUCTIONS/CHARACTER PASSED
LENGTH OF PATTERN
0
1
2
3
4
5
6
7
0
2
4
6
8
10
12
14
3.56
0.473
0.266
SA-4079-2

FIGURE 2

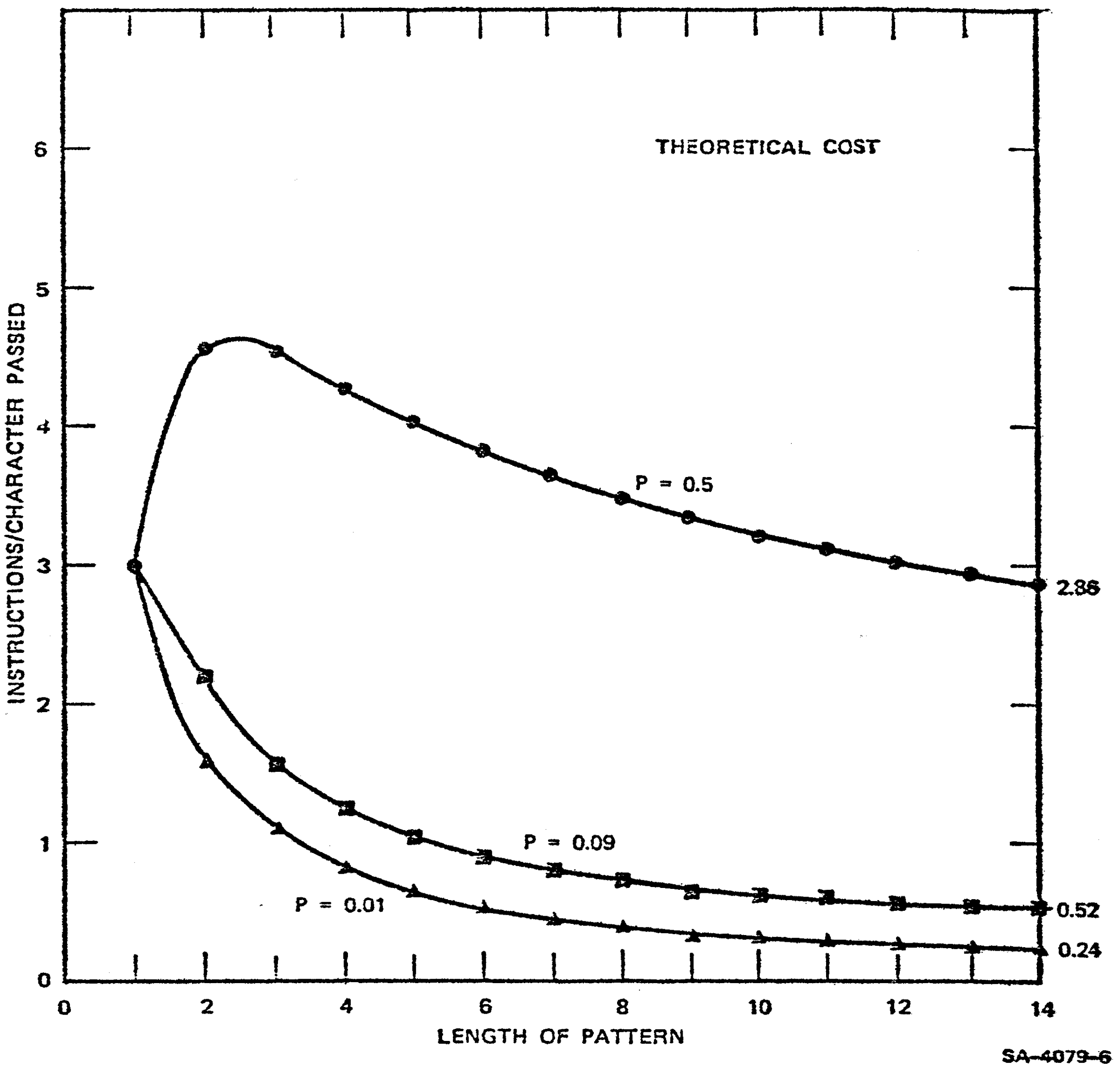
THEORETICAL COST
INSTRUCTIONS/CHARACTER PASSED
LENGTH OF PATTERN
P = 0.5
P = 0.09
P = 0.01
2.86
0.52
0.24
0
1
2
3
4
5
6
0
2
4
6
8
10
12
14
SA-4079-6

FIGURE 4