1991

String pattern matching algorithms: An empirical analysis

Edward J. Smith

The University of Montana

STRING PATTERN MATCHING ALGORITHMS: AN EMPIRICAL ANALYSIS

By

Edward J. Smith

B. A. Gonzaga University, 1976

Presented in partial fulfillment of the requirements

for the degree of

Master of Science

University of Montana

1991

Approved by

Ronald E. Wilson

Chairman, Board of Examiners

Dean, Graduate School

Aug. 13, 1991

Date

Smith, Edward J., M.S., August 1991 Computer Science

String Pattern Matching Algorithms: An Empirical Analysis

Director: Ronald E. Wilson REW

The problem of searching through text to find a specified substring, "pattern", is empirically examined. Several existing pattern matching algorithms are surveyed including the Knuth-Morris-Pratt and the Boyer-Moore algorithms as well as Daniel M. Sunday's algorithms. A technique of Boyer and Moore's, the fast loop, is extended to other algorithms with a dramatic improvement in performance. A short and simplified version of the Boyer-Moore algorithm is presented which is easy to understand and is very fast. Combining ideas from several different algorithms, a hybrid algorithm has been developed which maximizes the efficiency of the Boyer-Moore fast loop. This algorithm has excellent run time performance.

Algorithms which search strings of binary and quaternary alphabets are also presented. These algorithms process four and eight characters at a time by expanding a small sized alphabet into what ostensibly is a much larger alphabet.

Table of Contents

List of Tables

Acknowledgments

This research paper is dedicated to Dr. Ron Wilson whose work on nucleic acid sequence data manipulation served as the motivation for this paper. His expertise on this subject led to the expanded alphabet algorithms presented in this paper.

I would also like to dedicate the paper to Dr. Alden Wright whose help and inspiration was invaluable during all phases of my graduate studies.

Special thanks are due to my colleagues and friends Yu Shi and Jeff Heng who showed me more computer tricks than I can remember.

Finally, and most importantly, gratitude is due my father, Jack Smith, who taught me how to read. Without that, none of this would have been possible.

Introduction

Searching strings for patterns has long been a cornerstone of software methodology. So fundamental is the pattern matching task that most large programs employ it in one form or another. Pattern matching is in wide spread use in text editors, word processors, bibliographic search, data retrieval, symbol manipulation as well as specific applications such as nucleic acid sequence manipulation, speech recognition and the study of bird songs. The string searching field is continually expanding as scientists develop ways to encode their data so that it can be stored in a computer's memory and manipulated by computer programs. Some text strings exceed a million characters in length emphasizing the need for fast and efficient string searching algorithms.

The following paper examines and develops string searching algorithms with the intention of improving their performance. Several existing algorithms are discussed including the Knuth-Morris-Pratt (KMP) [3], the Boyer-Moore (BM) [1] and Daniel M. Sunday's [8] algorithms. Several variations of these algorithms are developed which exhibit improved performance. One variation of the BM algorithm, called str_search (ss), is easy to understand and code and is very fast. It is empirically shown that all Boyer-Moore algorithm versions are substantially faster than Sunday's algorithms when searching English text. This is in contradiction to Sunday's claims of superior performance for his algorithms.

The main improvement in the performance of all the algorithms studied is based on the 'fast loop', an idea originally presented by Boyer and Moore. This loop takes advantage of the right to left scan order of the BM algorithm thereby enabling the pattern to move rapidly along the text string whenever a mismatch occurs between the text character aligned with the character at the right end of the pattern and p[m-1]. Analogues of the 'fast loop' have been added to all the algorithms resulting in a dramatic improvement in speed and efficiency. Mysteriously, the 'fast loop' concept has been virtually ignored by researchers and text book authors since it was first introduced in 1977 by Boyer and Moore.

Combining some of the best ideas from separate algorithms, a hybrid algorithm referred to as least frequent character Boyer-Moore (LFBM), has been developed which is very fast and efficient and relatively simple to understand and code. This algorithm attempts to maximize the efficiency of the 'fast loop'.

Special case algorithms which search strings of binary and quaternary alphabets are given. These algorithms expand 2 and 4 character alphabets into a much larger alphabet enabling them to process 8 and 4 characters at a time respectively. Quaternary strings, i. e. strings over a 4 letter alphabet, are of special interest since they include DNA and RNA base sequence strings. These algorithms are based on the Boyer-Moore algorithm and have relatively fast run times.

Finally, any algorithm which is carefully and efficiently coded displays improved performance over many versions which appear in the literature.

Chapter One

Existing Algorithms

The string matching problem is defined as follows. Given two strings, the text string, t, and the pattern string, p, find the first occurrence of the pattern in the text. The pattern and text strings are considered to be arrays whose elements are characters. The text is indexed from 0..n-1 and the pattern is indexed from 0..m-1 and are of lengths n and m respectively with n >= m, (in practice, n >> m). When an occurrence of the pattern is found in the text, the text string index of the character which aligns with the first character of the pattern is returned. If no match is found, -1 is returned. The term "false start" will be used for situations where a pattern begins to match its current alignment by matching individual text characters but then encounters a mismatched character before completing the entire match.

1.1 Brute Force

The obvious string searching method, referred to as brute force (bf), searches for an occurrence of the pattern at each position in the text string. Initially, the pattern is aligned at the left end of the text so that text[0] is aligned with pattern[0]. If the first comparison of characters is successful, brute force proceeds rightward comparing characters until either a match is found or a false start is encountered. If the initial character comparison is a mismatch or a false start is encountered, the pattern is shifted one character to the right and the text string index is reset to align with the leftmost pattern character. This search scheme continues until a match is found or the text is exhausted.

In practice, a mismatch is usually detected at the first comparison position. However, in a situation where a pattern of 'aaaab' is searched for in a text of 'aa....ab', a worst case situation, brute force is clearly O(nm) [8]. This O(nm) worst case complexity results from the way the algorithm behaves when a false start is encountered. When this occurs, the index of the text string must be 'backed up' from the last comparison position of the false start to the text character which aligns with p[0]. For example, when trying to match the pattern 'aaab' in the text string 'aaaaaaab', a mismatch is first detected at t[3] where the text character 'a' is mismatched with the pattern character 'b'. The pattern is shifted right one character to check for the next potential match. To resume character comparisons, the text string index backs up to t[1] to compare text[1] with p[0]. Thus a total of n*m comparisons will be made before a match is found. The backing up of the text string index can lead to more serious inefficiencies if the whole text string is not available in memory and buffering operations are necessary. However, when searching English text, brute force usually does not exhibit this worst case behavior and its empirically determined running time is O(n) [8]. Taking into consideration the ease of coding and understanding, brute force is a good choice for searching short strings.

The brute force algorithm follows:

```
i := 0;         (*  text string index     *)
j := 0;         (*  pattern string index  *)

repeat
    if ( text[i] = pattern[j] )
        then begin  i := i + 1;  j := j + 1   end;
        else begin
                 i := i - j + 2;    (* back up! *)
                 j := 1
             end;
until ( j >= pattern_length )  or
                     ( i >= text_length );

if ( j = pattern_length )
    then                    (* found a match *)
        return ( i - pattern_length )
    else                    (* no match found *)
        return -1
```

1.2 Kunth-Morris-Pratt

In 1977, Knuth, Morris and Pratt (KMP) [3], sought to develop an algorithm which would avoid the brute force problem of backing up in the text string, i. e. decrementing the text string index. Once a comparison was made at t[i], for example, all subsequent comparisons would occur at positions >= i. This would hopefully yield an algorithm with a worst case running time of O(n).

In analyzing the brute force algorithm, Knuth, Morris and Pratt observed that when a false start occurs, information about the text string has already been collected. Possibly one can take advantage of this information to avoid backing up in the text string.

The key to the KMP algorithm is a table which stores values indicating how far to "slide" the pattern to the right after a mismatch is encountered. This table, referred to as the "next" table, can be referenced at the same position at which a mismatch occurs in the pattern. For example, if a mismatch occurs at p[j], the next table is referenced at next[j]. The next table value at this location indicates where the pattern should be checked next. The pattern is then aligned j-next[j] places to the right of the current alignment relative to the text.

The idea behind the next table involves shifting the pattern to the right in order to align already scanned and matched text with the nearest matching prefix of the pattern. When this shift occurs, it is also necessary to bring a different pattern character into the new alignment than the one which initially caused the mismatch. Once a new alignment is determined, comparisons resume at the point of the initial mismatch thereby avoiding any backtracking in the text string. This results in an O(n) worst case search time. The next table can be preprocessed from the pattern string in O(m) time giving an overall worst case performance for the KMP algorithm of O(n+m).

Next[j] is thus defined as the largest integer i < j such that p[0]...p[i-1] is a suffix of p[0]...p[j-1] and p[i] <> p[j].

Next[j] can be computed by the following algorithm:

```
next[0] := -1;

for j := 1 to pattern_length-1 do
    begin
        i := next[j-1];
        while( i >= 0 and pattern[i] <> pattern[j-1] )
            i := next[i];
        next[i] := (i+1)
    end;

(* this loop ensures that a different pattern
   char is brought into the mismatched position *)
for j := 1 to pattern_length-1 do
    if ( next[j] <> 0 ) and
                    ( pattern[j] = pattern[next[j]] )
        next[j] := next[next[j]];
```

Sedgewick [6] has devised a method to enable one to intuitively grasp the meaning of the next table. For any pattern string, one can construct a table and scan over prefixes of the pattern to determine the longest suffixes which match a prefix of these partial patterns. Thus for the pattern 'abcabcacab', we can construct the following table:

<table>
<tr><th>j</th><th>p[0..j-1]</th><th>f[j]</th></tr>
<tr><td>1</td><td>a|</td><td>0</td></tr>
<tr><td>2</td><td>ab|</td><td>0</td></tr>
<tr><td>3</td><td>abc|</td><td>0</td></tr>
<tr><td>4</td><td>abc|a</td><td>1</td></tr>
<tr><td>5</td><td>abc|ab</td><td>2</td></tr>
<tr><td>6</td><td>abc|abc</td><td>3</td></tr>
<tr><td>7</td><td>abc|abca</td><td>4</td></tr>
<tr><td>8</td><td>abcabcac|</td><td>0</td></tr>
<tr><td>9</td><td>abcabcac|a</td><td>1</td></tr>
</table>

F[0] is predefined as -1. Thus for every partial pattern p[0..j-1] of the pattern, we can check for a longest possible suffix which matches a prefix of that partial pattern. The length of this longest suffix is the 'f' tables value for the partial pattern p[0..j-1]. For example, for the pattern prefix at j = 7, 'abcabca', the longest suffix is 'abca' which is a prefix of 'abcabca'. This longest suffix is of length 4 so 4 is the 'f' value at that position, i. e. f[7] = 4.

This table however, does not take into account the fact that the new alignment must have a different pattern character brought into the position at the original point of mismatch. Otherwise, there will be another immediate mismatch. This problem is easily remedied by checking for each j whether p[j] = p[f[j]]. If so, the f[j] value must be changed to f[f[j]]. The resultant table would then be the 'next' table:

j =	0	1	2	3	4	5	6	7	8	9
next[j] =	-1	0	0	0	0	0	0	4	0	0

This adjustment is not necessary for the correct function of the algorithm, however, it does add efficiency to its operation.

The KMP algorithm improves worst case efficiency for the string search by never backing up in the text string. However, preprocessing the pattern to gain the 'next' shift does not add significant speed for searching English text over that of the brute force algorithm [6, 8]. For highly repetitive patterns, the KMP algorithm will perform more efficiently than brute force however, search time statistics are typically dominated by the event of a mismatch at the first character tested, p[0], [8]. In most applications, KMP and brute force perform about the same [2, 6, 7, 8].

The kmp algorithm follows:

```
i := 0;                 (* the text string    *)
j := 0;                 (* the pattern string *)

repeat
    if ( j = -1 )       (* had a mismatch at p[0] *)
        then begin   i := i + 1;  j := j + 1   end;

    if ( text[i] = pattern[j] )
        then begin   i := i + 1;  j := j + 1   end;
        else       (* (t[i] <> p[j]),  a mismatch *)
            j := next[j];
until ( j >=  pattern_length ) and
                    ( i >= text_length );

if ( j >= pattern_length )
    then                        (* found a match   *)
        return ( i - pattern_length )
    else                        (* no match found *)
        return -1;
```

1.3 Boyer-Moore

Early pattern matching algorithms aligned the pattern with the text and compared characters of the two strings in a intuitive left to right order. In 1977, Boyer and Moore discovered that more information could be gained about the text string if a right to left scan order of the pattern was used. In fact, using this protocol, many text characters can frequently be skipped without any comparisons leading Boyer and Moore to claim their algorithm to have "average" sub-linear complexity [1].

The Boyer-Moore algorithm initially aligns the pattern and text strings at their left but compares t[m-1] with p[m-1] first, then t[m-2] with p[m-2], etc., until either a match or a mismatch is found. Typically a mismatch occurs at the first comparison, t[m-1]. If character t[m-1] does not occur in the pattern string, the pattern can be shifted m characters to the right so that t[m] is aligned with p[0]. If the character t[m-1] does occur in the pattern, the pattern is shifted right so that character t[m-1] is aligned with the first right most occurrence of that pattern character. This heuristic is known as the delta1 shift. Values for every character in the string alphabet are contained in the array, delta1. Values for delta1 are determined by a precomputation of the pattern where a delta1 value is the distance a character is located from the end of the pattern string. For example, for the pattern string 'zipper', delta1[r] = 0, delta1[e] = 1, delta1[p] = 2, delta[i] = 4, etc. If a character does not occur in the pattern, then its delta1 value is m.

The Boyer-Moore algorithm uses another precomputed table, delta2, to help maximize shifts of the pattern after a mismatch beyond the first comparison has occurred. This idea, analogous to Knuth, Morrison and Pratt's 'next' table, is computed by taking the already matched suffix of the pattern, p, to the right of the first mismatched character and by finding the next leftward occurrence of it in p. Sliding the left occurring suffix into the position of the already matched one gives a new alignment however, the character at the mismatch position must be different from the original mismatched pattern character. Both delta1 and delta2 values are functions of where the mismatch occurs in the pattern. The new alignment is determined by incrementing the text string index by the maximum of the delta1 and delta2 shift values.

The preprocessing of the pattern string to construct the delta1 and delta2 tables takes O(m) time. Therefore, the Boyer-Moore algorithm has a worst case complexity of O(n+m). The Boyer-Moore algorithm follows:

```
i := pattern_length - 1;          (* init text index *)

while ( i < text_length )
    begin
        j := pattern_length - 1;
        while ( text[i] = pat[j] )
            begin
                i := i - 1;
                j := j - 1;
                if ( j < 0 )      (* found a match *)
                    return (i + 1)
            end;
        i := i + max( delta1[ text[i] ], delta2[j] )
    end;

return ( -1 );                    (* no match found  *)
```

1.4 Boyer-Moore Fast Loop

Once Boyer-Moore [1] established the heuristics for their search algorithm, they began looking for ways to further improve its performance. They observed that whenever a new alignment is determined, there is about a 93% chance of a mismatch at the first comparison using English text. To take advantage of this high probability of mismatch, they coded their algorithm so that in this most frequent case, the algorithm stays in a simple loop that is coded with a small number of machine instructions. They called this loop the 'fast loop'.

The main idea of the fast loop is to scan down the text string incrementing the text index by the value delta1['text'] where 'text' is the text string character opposite p[m-1]. However, instead of using delta1, Boyer-Moore defined a new table, delta0, which contains the same values as delta1 except that delta0[pattern[m-1]] is set to integer 'large' which is greater than text length (n) + pattern length (m). (Recall delta1[pattern[m-1]] is always 0). When a match opposite p[m-1] occurs, the text string index is incremented by the large value. At this point, the text string index exceeds the text string length. This condition serves as the loop terminator. Control can also leave the loop when the text string is exhausted indicating no match was found. A test follows the termination of the loop which determines whether the text actually is exhausted or whether a hit with pattern[m-1] occurred. If a hit has occurred, then the text index is restored by subtracting the large value from the text index and comparisons proceed.

While in the fast loop, a short pattern is typically shifted m characters for most loop iterations. Boyer-Moore estimate that 80 percent of the search time is spent in this short loop. As a result, a majority of the text string remains unexamined leading Boyer-Moore to claim their algorithm to be sub-linear [1].

The following code segment is taken directly from the Boyer-Moore paper [1] and illustrates the ’fast’ loop:

```
        i <-- pattern length - 1
fast:   i <-- i + delta0[text[i]]
        if ( i <= text length ) then goto fast
```

The Boyer-Moore algorithm with a variation of the fast loop follows. The array name delta1 is used instead of the Boyer-Moore name delta0.

```
i := pattern_length - 1;     (* init text index    *)

if ( pattern_length > text_length )
    return -1;               (* error check input *)

while ( true )
    begin
        repeat               (* ’fast’ loop   *)
            i := i + delta1[ text[i] ];
        until ( i >= text_length );

        if (i < delta1[ pattern[pattern_length-1] ])
                           (* text string expired  *)
            return -1;     (* no match found  *)

        (* reset string indexes *)
        i := i - delta1[ pattern[pattern_length-1] ];
        j := pattern_length - 1;

        while ( text[i] = pattern[j] ) do
            begin
                if ( j = 0 )
                    return i;     (* match found  *)
                i := i - 1;
                j := j - 1
            end

        if ( delta1[ text[i] ] < delta2[j] ) or
                 ( delta1[ text[i] ] > pattern_length )
                then i := i + delta2[j]
                else i := i + delta1[ text[i] ]
          end;  (* while ( true ) *)
```

1.5 Sunday's Algorithms

In August, 1990, Daniel Sunday published a paper entitled: "A very fast substring search algorithm" [8]. In his paper, Sunday actually presents three algorithms all of which he claims to out perform the Boyer-Moore algorithm. They are: quick search, maximal shift and optimal mismatch.

Unlike the Brute Force (BF), the Knuth-Morris-Pratt (KMP) and the Boyer-Moore (BM) algorithms, Sunday's algorithms do not require the pattern to be searched in any particular order. One can use a right to left search order like the BM algorithm or a left to right order like KMP or any other order. The scan order is stored in an array of records 'pattern[]'. Each record contains two fields: location and character. The location field stores the actual index of the character in the pattern string. Thus, for a BM scanning of the pattern, 'string', the character 'g' would have location 5 but would be stored as the 0th element in the array of records, each record of type 'PAT'.

Sunday's algorithms rely heavily on the BM and the KMP algorithms for its searching conventions. Two tables are defined: TD1 which is analogous to BM's delta1 table and TD2 which is analogous to BM's delta2 and KMP's next tables. Both tables are created by preprocessing the pattern string.

TD1 is computed by taking the distance of each pattern character from the end of the pattern plus one. For example, for the pattern 'string', TD1[g] = 1, TD1[n] = 2 etc. TD1 values are 1 greater than BM delta1 values because when a mismatch is encountered, the algorithm relies on the first text character past the current alignment for the TD1 index and not the pattern character which caused the mismatch. Since the TD1 table is referenced by a character outside of the current alignment, the TD1 shift is independent of the order in which the pattern is scanned.

The second table, TD2, is a function of the position in p where a mismatch first occurs. TD2 is calculated after a scan order of the pattern is determined. When a mismatch is encountered, it is necessary that the TD2 shift give an alignment such that a different pattern character than the one which caused the mismatch be brought into the corresponding text location. Otherwise, there would be another mismatch. For an ordering of the pattern I[], TD2[j] is defined as the minimum left shift such that p[I[0]]...p[I[j-1]] match their aligned characters in the pattern string but p[I[j]] does not. TD2[0] is predefined as 1.

If the pattern string is scanned left to right, then TD2 is KMP's next table. If the pattern string is scanned right to left, then TD2 is similar to the BM delta2 table. For a right to left scan order, TD2 values do not match BM delta2 values as one might intuitively expect. However, this discrepancy is easily reconciled by examining how each algorithm handles the pointer or index into the text string. In Sunday's search algorithm, the text pointer, tx, points to the position in the text which aligns with p[0]. As comparisons proceed, 'tx' is not incremented, rather the location field of the 'PAT' record is used as an offset to check if individual characters match. If a false start is discovered, tx is then incremented to point to the start of the new alignment where the next check for a potential match will occur. On the other hand, BM's text string index initially points to the text character which is aligned with p[m-1] and is decremented for each successful comparison. When a false start occurs, it is incremented by the delta2 value corresponding to the location in the pattern of the mismatched character. If these two very different protocols are taken into account, the TD2 table and BM's delta2 table are identical for the right to left scan order. An analogous situation exists for a left to right scan order with KMP's next table being identical to TD2.

As mentioned earlier, Sunday has developed three algorithms: quick search (qs), maximal shift (ms) and optimal mismatch (om). Each algorithm is based on a different scan ordering. Quick search is a brute force/Boyer-Moore hybrid. It uses brute force's conventional left to right scan order and the BM delta1 analogue, TD1. Optimal mismatch and maximal shift actually use the same search algorithm, however, they both use an unconventional pattern scan ordering and the two algorithms are quite different.

1.5.1 Quick Search

Quick search is basically the brute force algorithm with the TD1 shift which adds considerable efficiency to the search. Both pattern and text are initially aligned at the left end of the strings (i. e., p[0] aligns with t[0]) and scanning proceeds left to right. The pattern index is incremented for every successful comparison and also is used as an offset with the text string index which is only incremented after a false start. When a false start is encountered, the TD1 look up references the text character 1 character past the current alignment. This value is then added to the text string index to give a new alignment which, in turn, is checked for a match. The quick search algorithm follows, [8].

```
got_match := false;
k := 0;


while ( got_match = false ) and ( k + m <= n ) do
    begin
        i := 0;       (* init pat index for l-r scan *)
        while ( i < m ) and ( p[i] = text[k+i] ) do
            i := i + 1;       (* increment pat index *)
        if ( i = m )          (* all pat chars match *)
            then
                got_match := true
            else           (* false start, shift pat *)
                k := k + TD1[text[k+m]]
    end;
```

```
if ( got_match = true )
    then QSearch := k       (* match found at text[k] *)
    else QSearch := -1      (* no match found         *)
```

Sunday suggests that quick search can be improved by adding the TD2 shift to the algorithm. This algorithm, referred to as quick search 2, (qs2), uses the pattern string index at the place of mismatch for the index into the TD2 array. The maximum of the TD1 and TD2 shifts is then added to the text string index. Any advantage gained in performance in using a TD2 shift may be offset by the time lost in computing TD2.

1.5.2 Maximal Shift

The idea behind the maximal shift (ms) algorithm is to maximize the TD2 shifts. A pattern scan ordering is created where the pattern character whose next leftward occurrence is a maximal distance is checked against its aligned text character first. Of the pattern characters remaining, the character whose next leftward occurrence is a maximum distance is checked second etc. If a false start is encountered at a subsequent comparison, then the TD2 shift will be maximized.

The ordering is computed by first calculating an array, 'Minshift', which holds the leftward distance to the next occurrence for each character in the pattern. For example, for the pattern: 'abcabcacdab', the Minshift array will contain the values: 1 2 3 3 3 3 3 2 9 3 6. The Minshift array is then sorted into descending order and the corresponding pattern characters and their locations in the pattern string are stored in the array of records, 'pattern'. This array will contain the values:

CHAR	d	b	a	a	c	b	a	c	c	b	a
LOC	8	10	9	6	5	4	3	2	7	1	0

Checking the character 'd' first, will give a maximal shift if there is a mismatch past this character. Suppose the 'd' character at position 8 in the pattern matches the corresponding text character. As further comparisons proceed, if any mismatch is encountered, the pattern can be shifted 9 positions before the possibility of a 'd' character in the text comes into a position to match the 'd' of the pattern. Notice that if two pattern characters have the same leftward shift, the BM scan order is used and the right most character is checked first.

1.5.3 Optimal Mismatch

The idea behind the ordering of the optimal mismatch (om) algorithm is to compare characters in the pattern which occur the least frequently in English text first. This increases the probability of finding a mismatch early in text-pattern comparisons and results in greater efficiency. The ordering of the pattern is easily calculated by referencing a table of alphabet frequencies for English text. Characters are arranged in order of increasing frequency. For example, for the pattern string 'extraordinary', the ordering would be:

CHAR	x	y	d	n	o	t	r	r	r	i	a	a	e
LOC	1	12	7	9	5	2	11	6	3	8	10	4	0

Note, if two or more of the same letter occurs in the pattern, the BM scan order is applied and the right most occurrence is checked first.

Checking the character least likely to occur first can dramatically improve performance. The most frequently occurring character, 'e', has about a 10 percent occurrence rate. Furthermore, about 20 percent of English words end in the letter 'e' [8]. Using the BM right to left scan order increases the likelihood of a match at this first comparison whereas using the om heuristic minimizes this probability. Sunday has estimated that the average ratio of the text occurrence probability of the last letter of a word to the least likely letter in it is almost 5. Therefore, if the least likely letter of a word is tested first, it is 5 times more probable that it will produce a mismatch.

The search algorithm for the maximal shift and the optimal mismatch scan orders follows.

```
got_match := false;
k := 0;

while ( got_match = false ) and ( k + m <= n ) do
    begin
        j := 0;          (* init pat index for scan *)
        while ( j < m ) and
```

```
                        ( p[I[j]] = text[k+I[j]] ) do
            j := j + 1;    (* increm pat index    *)
        if ( j = m )       (* all pat chars match *)
            then got_match := true
            else           (* false start         *)
                begin      (* shift pattern       *)
                    delta1 := TD1[text[k+m]];
                    delta2 := TD2[j];
                    k := k + max(delta1,delta2);
                end;
    end;

    if ( got_match = true )
        then QSearch := k       (* match found    *)
        else QSearch := -1      (* no match found *)
```

1.6 Scan Least Frequent Character

When searching English text for an occurrence of a pattern, it may increase search efficiency to check the text first for an occurrence of the least frequently used English text character in the pattern. For example, if the pattern is the English word 'extra', it would be most efficient to check the text for an occurrence of the character 'x' first. Brute force looks first for an occurrence of the character 'e' which is the most commonly used letter in English text, occurring once about every ten characters. The character 'x', however, occurs about once in every 33 letters [8]. This increases the likelihood of mismatches at the first comparison therefore decreasing the number of false starts which improves the efficiency of the algorithm.

This algorithm, scan least frequent character (slfc), first preprocesses the pattern string to determine the least frequently used character as well as its offset from the start of the pattern, p[0]. This least frequent character is then concatenated onto the end of the text string at text[text_length] to prevent the text string index from running off the end of the text in case a match is not found. With these preliminaries aside, the algorithm enters a loop which scans the text string left to right searching for an occurrence of the least frequent pattern character. When such an occurrence is located, the loop is terminated and a check for end of text is made. If text still remains, comparisons proceed left to right until either a match is found or a false start is detected. If a false start is encountered, the text index is incremented by one and the search resumes by reentering the loop which scans the text string for the least frequent character. The only added information needed for the algorithm is a table of English text alphabet frequencies. This is easily computed by running an information gathering program on a large volume of English text. Such a table is given in the appendix.

The scan for least frequent character algorithm which follows is based on a similar algorithm by Horsepool [2].

```
if ( pattern_length > text_length )
    return -1;        (* failed to match *)

least_freq := 25.0;  (* 25.0, an init dummy value *)

(* find least frequent char *)
for j := 0 to pattern_length-1 do
    if ( Freq[ pattern[j] ] < least_freq )
        begin
            least_freq := Freq[ pattern[j] ];
            char := pat[j];  (* store lfc in char *)
            k := j    (* store offset of lfc in k *)
        end;

i := k;                         (* initialize text index *)
text[text_length] := char;   (* cat char onto text *)

repeat
    while ( text[i] <> char )    (* search for lfc *)
        i := i + 1;

    if ( i >= text_length )     (* match not found *)
        return -1;

    j := 0;                       (* init pat pointer *)
    temp_i := i;                  (* store text index *)
    while ( text[i - k] = pattern[j] ) and
                              ( j < pattern_length ) do
        begin
            i := i + 1;
            j := j + 1
        end;

    if ( j = pattern_length )   (* found a match    *)
        then  return (i - k - pattern_length);

    i := temp_i + 1;     (* continue search at i+1 *)
until ( true );
```

Chapter Two

Algorithm Refinements

2.1 The Ishift Loop

In their original paper, Boyer and Moore showed that the performance of their algorithm could be improved dramatically when implemented with the fast loop. Therefore, a loop analogous to the fast loop has been used to improve the performance of a variety of algorithms. This loop is referred to as the 'ishift' loop. The loop is essentially a reworking of the 'fast' loop, the only difference being the absence of the 'large' value. Algorithms which use the ishift loop keep the value 0 at the delta1[pattern[m-1]] position. The loop condition then becomes: while (ishift <> 0). The ishift value must be primed before the loop condition is initially tested. Once the loop is entered, the text index is incremented by the ishift value. A new ishift value is then fetched from delta1 using the text character opposite p[m-1] as the delta1 array index. An example of the ishift loop follows:

```
i := pattern_length - 1;          (* init text index *)
ishift := delta1[ text[i] ];      (* prime ishift    *)

while ( ishift <> 0 )             (* the ishift loop *)
    begin
        i := i + ishift;
        ishift := delta1[ text[i] ];
    end;
```

By omitting the large value, it is now possible to increment the text index past the end of the text string if a match is not present. To offset this problem, the pattern can initially be concatenated onto the end of text string and when matches are found past the actual text length, a failure is reported. Another way to deal with this problem is to add another condition to the ishift loop: while (ishift <> 0 AND text_index <= text_length). However this solution is not the most efficient because it adds another condition to a part of the algorithm which is frequently executed. Yet another way to cope with the problem of going off the end of the text is to concatenate null values on the end of the text string. These null values would then have corresponding values of -1 in the delta1 array. The ishift loop condition now becomes: while (ishift > 0).

The ishift loop can be employed in any algorithm which uses a delta1 or TD1 table. Even if a delta1 table is not used, an analogous idea can be employed to take advantage of mismatching at the first character position. For example, in the KMP and BF search, p[0], the first character of the pattern can be checked against the present text position. While these two characters are not equal, the text index can be incremented. As with the ishift loop, care must be taken to not "run off" the end of the text string. The ishift analogue for the KMP and BF algorithms follows:

```
        first_pattern_char := pattern[0];
        i := 0;
```

```
while ( text[i] <> first_pattern_char ) do
    i := i + 1;
```

These modifications greatly enhance the performance of the algorithms. When the loop is executing, the text string index is rapidly moving down the text string scanning for a hit with the appropriate pattern character. As long as this pattern character is not found, the text index continues to be incremented. Only when the target pattern character is found do text-pattern comparisons actually take place.

2.2 The Str_Search (ss) Algorithm

To quickly and easily code a fast string search algorithm, the BM algorithm with one delta table, delta1, can be implemented with the 'ishift loop' to give very good overall performance. No delta2 table is used.

This algorithm concatenates the pattern onto the end of the text string to prevent running off the end of the text while in the ishift loop. If a character match occurs between the last pattern character and the aligned text character, the ishift loop is exited and comparisons proceed in the right to left BM order. If a false start is encountered, the text string index is incremented by the maximum of delta1[text[i]] and m - j where i and j are the text and pattern string indexes of the characters which caused the mismatch.

The ss version of the BM algorithm is kept simple and short to facilitate ease of coding and understanding while at the same time it retains over all performance as good as any other algorithm.

The ss algorithm follows:

```
j := 0;

(* concat pattern onto text *)
for  i := text_length to ( text_length +
                            pattern_length - 1 ) do
    begin
        text[i] := pattern[j];
        j := j + 1
    end;

i := pattern_length - 1;   (* init text string index *)
while ( i < text_length )  (* while more text remains*)
    begin
        ishift := delta1[ text[i] ];
        while ( ishift <> 0 )    (* ishift/fast loop *)
            begin
                i := i + ishift;
                ishift := delta1[ text[i] ];
            end

        if( i < text_length )    (* text still left? *)
            begin
                j := pattern_length - 1;
                repeat         (* make r-1 comparison*)
                    i := i - 1;
                    j := j - 1;
                until ( j < 0 ) or
                             ( text[i] <> pattern[j] );
                if( j < 0 )
                    then      (* match found *)
                        return i + 1;
                    else
                     (* false start; incr text index *)
                        if ( delta1[ text[i] ] >
                                   pattern_length - j )
                            then i := i +
                                     delta1[ text[i] ];
                            else i := i +
                                    pattern_length - j;
            end (* if *)
    end (* while *)
return (-1);              (* failed to match  *)
```

2.3 A Fast Loop Analogue For Sunday's Algorithms

An analogue of the BM fast loop has been added to the three algorithms developed by Sunday [8]. The addition of this loop shows a marked improvement in the algorithms performance. All three algorithms have been altered in the same way by the addition of the following lines of code:

```
i := 0;
j := 0;

while ( text[i] <> pattern[j] ) do
    i := i + TD1[ text[i + pattern_length] ];
```

This loop, like Boyer-Moore's fast loop, takes advantage of the likelihood of a mismatch at the first comparison location. It therefore checks that the characters do indeed mismatch and if they do, the text index is incremented by the TD1 value. This value will usually increment the text index by m allowing much of the text string to be skipped over without any comparisons whatsoever. Only when the text character aligned with the first pattern character match do further comparisons take place.

Chapter Three

Other Algorithms

3.1 Least Frequent Character/Boyer-Moore Hybrid

In an attempt to create even faster more efficient string searching algorithms, some of the better ideas from two of the algorithms have been synthesized. The least frequent/Boyer-Moore algorithm (lfbm) is essentially a reworking of the Boyer-Moore algorithm with one delta table, delta1. The algorithm uses the 'ishift' loop to scan down the text string as efficiently as possible. If the loop terminating condition is encountered, (ishift = 0), the first character comparison made is the character in the pattern which occurs least frequently in English text. If this comparison is a mismatch, the text string index is incremented and the ishift loop is immediately reentered. If the least frequent character comparison is a match, comparisons then proceed in a right to left, Boyer-Moore fashion until either a match is found or a mismatch is detected. If the condition is a false start, the text index is incremented by the delta1 value fetched from delta1[j] where j is the pattern string index of the character which caused the mismatch.

This algorithm attempts to discover false starts as early as possible, thus avoiding needless comparisons. As soon as a single character match at the right end of the current alignment is detected, (i. e. ishift = 0), the algorithm then checks if the text character opposite the least frequent pattern character matches. If the characters match, there is a high probability the current alignment will yield a match thus avoiding a false start. If there is a mismatch after this single comparison, the ishift loop is immediately reentered and the search for the next potential match alignment resumes.

The scan least frequent character/Boyer-Moore algorithm follows:

```
(* error check input *)
if ( pattern_length > text_length )
    return -1;

j := 0;                (* concat pattern onto text *)
   for i := text_length to
              ( text_length + pattern_length - 1 ) do
    begin
        text[i] := pattern[j];
        j := j + 1
    end;

temp_freq := 25.0;     (* init temp_freq w/ dummy  *)

(* find least freq char *)
for j := 0 to pattern_length - 1 do
    if ( Freq[ pat[j] ] < temp_freq )
        begin
            temp_freq := Freq[ pat[j] ];
            k := j       (* store offset from p[0] *)
            least_freq_char := pat[j];
        end;

(* store offset from p[pattern_length-1]  *)
from_pattern_end := pattern_length - k - 1;

while ( i < text_length ) do
    begin
        repeat
            repeat       (* ishift loop  *)
             ishift := deltal[i];
             i := i + ishift;

            (* until t[i] = p[plen-1] *)
            until ( ishift = 0 );
            (* check if least freq char matches *)
            if ( text[i - from_pat_end] =
                                      least_freq_char )
                begin
                    temp_i := i;  (* store text index*)
                    break;
                end;

            i := i + delta1[i+1]; (* reset text index*)
        until ( true )  (* leave loop via break only *)

        if ( i >= text_length ) (* gone off end text *)
            return -1;          (* no match found    *)

        j := pattern_length - 1;   (* init pat index *)

        repeat          (* make right to left compar *)
            i := i - 1;
            j := j - 1;
        until (j < 0) or ( text[i] <> pattern[j] );

        if ( j < 0 ) then           (* found a match *)
            return ( i + 1 )
        else                        (* a false start *)
            begin
                i := temp_i;  (* retrieve text index *)
                i := i + delta1[i+1];
            end;
    end;
```

3.2 Expanded Alphabet

Not all string searching tasks involve looking for English words in English language text. In nucleic acid sequence manipulation, for example, pattern strings consisting of only four characters, A, C, G, and T, are searched for in text of the same alphabet size. Another useful string searching application involves searching binary strings. Both these applications involve small alphabets. The following is a discussion of the Boyer-Moore algorithm applied to these string searching problems. The BM algorithm has been modified to add efficiency to the string search when a small sized alphabet string is searched.

3.2.1 Quaternary Alphabet

When searching strings composed of a small alphabet, the chances of a mismatch at the first comparison are greatly reduced, thus reducing the amount of text the Boyer-Moore algorithm can skip over. When working with a four character alphabet: {A, C, G, T}, one can easily expand the alphabet into a 256 character alphabet by examining four characters at a time. With 'A' encoded as 00, 'C' as 01, 'G' as 10 and 'T' as 11, four character chunks of the text can be encoded into an eight bit binary word which indexes an array with bounds 0..255. The Boyer-Moore delta1 and delta2 tables can be used along with a third table, delta3. The delta3 shift handles cases where some or all of the first three characters of the pattern occur in the mismatched portion of the text.

The delta1 array is built by encoding four character chunks of the pattern and using these encodings as the indexes into the array. These four character chunks will be referred to as 'quadruplets'. The actual delta1 values are determined the same way they are determined in the 'traditional' Boyer-Moore algorithm, i. e. by computing the distance from the right end of the quadruplet to the end of the pattern. For example, the right most quadruplet, p[m-4]..p[m-1] has a delta1 value of 0. The next quadruplet to the left, p[m-5] to p[m-2] has a delta1 value of 1 etc. The leftmost quadruplet, p[0] to p[3], has as its value the distance from p[3] to p[m-1]. All other quadruplets which do not exist in the pattern take a delta1 value of m.

Since the delta1 array is indexed with quadruplets only, it is necessary to define a delta3 shift for the three characters at the far left of the pattern: p[0], p[1], p[2]. For example, for a pattern beginning with 'acg.....' of length 10, delta3[aacg] = delta3[cacg] = delta3[gacg] = delta3[tacg] = 7. Using the same example, delta3[aaac] = delta3[acac] = delta3[agac] = delta3[atac] = delta3[caac] = ... = delta3[ttac] = 8, etc. Thus for every {A, C, G, T} permutation of length 3 followed by p[0], there is a delta3 entry for that quadruplet equal to m-1. For every permutation of length 2 followed by p[0], p[1], there is a delta3 entry for that quadruplet equal to m-2 etc.

A delta4 table has also been defined which is minimum(delta1, delta3) to streamline the search. Now, if the following alignment is encountered, a reference to the delta4 array will provide the proper shift.

```
text:       ...ttacgt
pattern:       acgt
```

The text string quadruplet 'ttac' does not exist in the pattern, therefore its delta1 value is m which in this case is 4. For the same text substring, there is a delta3 entry of 2. Then minimum(4, 2) = 2 which is the proper shift and the correct match will be found.

With this bookkeeping aside, one can preserve the efficiency of the Boyer-Moore algorithm by coding the 'fast' loop which enables the algorithm to skip over text without doing comparisons. However, instead of using a 'large' value, the delta1 table remains unaltered and the value of 0 at delta1[p[m-4]..p[m-1]] serves as the loop terminator. To prime the loop, the right most four characters of the text which line up with the pattern are encoded. Using this encoding as an index into delta4, the delta4 value is fetched and the text index is incremented by that value. When this shift value becomes 0 or the text is exhausted, the loop is exited. If text remains and the shift value equals 0, there is no need to check the current quadruplet for a match since the delta1 value is 0. Comparisons then begin four characters to the left of the current text index. Comparisons proceed leftward until either a match is found or a mismatch is discovered. If a mismatch is encountered, delta2 is referenced using the pattern string index of where the mismatch occurred and that delta2 value is added to the current text index. Delta1 is not referenced because the lookup requires a four character encoding. We therefore settle for the delta2 shift in place of the Boyer-Moore maximum(delta1, delta2). The search is then resumed with this new text index. The algorithm for the quaternary alphabet string search follows:

```
i := pattern_length - 1;        (* init text index, i *)

while ( true )
    begin
        if ( i > text_length )      (* end of text?   *)
        return -1;             (*  then no match *)
    encoded_4 := 0;            (* init encoded_4 *)
    encoded_4 := encoding of
                        text[i - 3]..text[i];

    ishift := delta4[encoded_4];  (* init ishift *)
    while ( ishift > 0 and i < text_length )
        begin       (* BM's 'fast' loop analogue *)
            i := i + ishift;
            encoded_4 := 0;
            encoded_4 := encoding of
                            text[i - 3]..text[i];
            ishift := delta4[encoded_4];
        end;

    i := i - 4;         (* since ishift = 0 then *)
    j := pat_length - 5;   (* last 4 chars match *)

    while ( true )
        begin
            if ( j = -1 )
                return ( i + 1 ); (* match found *)
            if ( text[i] = pattern[j] ) then
                begin
                    j := j - 1;
                    i := i - 1;
                end;
            else break;         (* a false start *)
        end;

    i := i + delta2[j];  (* increment text index *)
end;
```

3.2.2 Binary Alphabet

Using the same methods of the expanded alphabet algorithm for a four character alphabet, an algorithm which expands the binary alphabet: {A, T} has been encoded. The characters A, T have been chosen for the binary alphabet instead of the traditional {0, 1} to facilitate comparisons with the A, C, G, T algorithm. Random strings composed of {A, T} can be generated [4] and both the quaternary and the binary alphabet algorithms can be executed using these strings as input.

The binary algorithm encodes A as 0 and T as 1 and examines eight character chunks at a time making an alphabet of size 256. This expanded binary alphabet algorithm is essentially a two character reworking of the quaternary algorithm described above.

3.3 Sunday's Quick Search With Two Tables

In Sunday's [8] discussion of his quick search (qs) algorithm, he mentions that performance may be enhanced if one augmented the algorithm with the TD2 table. A quick search algorithm with two tables has been implemented and is called quick search2, (qs2). The TD2 shift is employed only when a false start is encountered. When this condition occurs, the text index is incremented by the maximum of TD1[text[i+m]] and TD2[j] where i is the text index opposite p[0] and j is the pattern index of the pattern character which caused the mismatch.

This algorithm, along with Sunday's original quick search, are well suited to illustrate the tradeoffs of the disadvantage of taking the additional time to preprocess the pattern to make the TD2 table versus the advantage gained during the search of having the TD2 shift.

Chapter Four

Testing

4.1 Test Driver

To test each algorithm, a driver program was constructed and coded in C to call each search function and collect statistics on the algorithm's performance. These statistics include: the location of the pattern in the text string or -1 if the search failed, the time elapsed for the algorithm to complete the search, the number of times the text string was accessed and the total amount of text the search passed in order to complete the search. After having executed the algorithms, the driver outputs all the statistics pertaining to the search. The driver also has exception handling built in to flag conditions where one algorithm reports a different result than another. As new algorithms are developed and 'plugged in' to the driver, this exception handling capacity greatly aids the debugging process.

To invoke all the algorithms from within a common programming block, an array of pointers was set up containing a pointer to each search function. Each algorithm resides in a separate file and has a common interface. That interface contains four parameters: the text string, the text length, the pattern string and the pattern length. When text accesses are tallied, two more parameters are added to the interface: text accessed and the total amount of text involved in the search. All algorithms call subsidiary functions, such as pattern preprocessors, from within their own module. Search functions are declared as type 'integer' and return a -1 for a failed search or the text string index of where a successful search originated.

For each iteration through a standard 'for' loop, a different search function is called. Another loop is embedded within the outer loop which allows the driver to search a text file for several patterns sequentially. For each iteration through the inner loop, a new pattern is fetched and the current search algorithm is invoked. The patterns are stored in a separate file and must be separated by carriage returns. This allows each search algorithm to search a text file for as many patterns as desired. The total amount of time taken to find all pattern (non)-occurrences is stored for each algorithm as well as cumulative text accesses to total text ratios.

When any search function is added or deleted to/from the driver, only three changes need be made. A constant, NUM_ALGS, must be altered to reflect the new number of search algorithms to be executed. The name of the search function(s) must be added to the function pointer array and a string, an abbreviation of the algorithm name, to be printed out when the driver terminates, must be added to an array of function name strings.

The algorithm for invoking the search functions and gathering all statistics follows:

```
for  i := 1 to NUMBER_ALGORITHMS do
    begin
        total_time := 0.0;
        for k := 1 to number_patterns do
            begin
                get_next_pattern(next_pattern);
                Plen := strlen(next_pattern);
                total_text := text_accessed := 0;

                if  ( i = 1 )  then
                    begin      (* first alg run *)
                        before_time := get_time();
                        /* str search function call */
                        answers[k] := fn_ptr_array[i]
                             ( text, Tlen, pat, Plen );
                        after_time := get_time();
                    end
                else   (* i > 1, all other algs *)
                    begin
                        before_time := get_time();
                        answer := fn_ptr_array[i]
                             ( text, Tlen, pat, Plen );
                        after_time := get_time();
                        /* error check; same results?*/
                        if answer <> answers[k]
                              print  "*****error*****";
                    end

                (* accumulate run time statistics *)
                total_time := total_time +
                              after_time - before_time;
                frac_totals[ Plen ][i] :=
                        frac _totals[ Plen ][i] +
                            text_accessed / total_text;
            end   (* inner loop *)

            time[i] := total_time;
    end   (* outer loop *)
```

4.2 Test Methodology

Test runs were performed on all algorithms using a DECStation 5500 RISC machine. The test file used for the English text algorithms is the file 'words' found in the

"/usr/dict" directory on UNIX systems. It contains all words used on the UNIX system for spelling checks. The text string consisted of the entire 'words' file concatenated together. The patterns searched for were each individual word from the 'words' file. Random text and pattern strings were generated to test the expanded alphabet algorithms [4].

All algorithms were implemented in the programming language 'C', (a listing of the algorithms is given in the appendix). Three versions of each algorithm were tested: an array version and a pointer version which were both timed, and an untimed pointer version which counted text accesses.

Two statistics were collected to judge the performance of algorithms: time of execution, in 1/60ths second, and text string accesses versus total text passed in the search. The time of execution is machine dependent while the amount each algorithm accesses the text string in machine independent. Text accesses were counted whenever a comparison between a pattern character and text character occurred as well as any time a text character was used in a delta1 array look up. Listing 4 in the appendix contains a C code function version which illustrates how the text accesses and total text variables were counted.

The results of these tests follow.

4.3 Test Results

Table 4.1. Key to algorithm acronyms

ACGT:	expanded/quaternary alphabet, 1 delta table
ACGT2:	expanded/quaternary alphabet, 2 delta tables
BF:	brute force
BFI:	brute force with ishift loop analogue
BIN:	expanded/binary alphabet, 1 delta table
BIN2:	expanded/binary alphabet, 2 delta tables
BM1F:	Boyer-Moore, 1 delta table, fast loop
BM1S:	Boyer-Moore, 1 delta table, no fast loop
BM2F:	Boyer-Moore, 2 delta tables, fast loop
BM2S:	Boyer-Moore, 2 delta tables, no fast loop
KMP:	Knuth-Morris-Pratt
KMPI:	Knuth-Morris-Pratt with ishift loop analogue
LFBM:	least frequent Boyer-Moore
MS:	maximal shift
MSI1:	maximal shift, 1 delta table, ishift loop
MSI2:	maximal shift, 2 delta tables, ishift loop
OM:	optimal mismatch
OMI1:	optimal mismatch, 1 delta table, ishift loop
OMI2:	optimal mismatch, 2 delta tables, ishift loop
QS:	quick search
QSI1:	quick search, 1 delta table, ishift loop
QSI2:	quick search, 2 delta tables, ishift loop
QS2:	Sunday's quick search with 2 delta tables
SLFC:	scan least frequent character
SS:	string search; (simplified Boyer-Moore)

Table 4.2. Timed results, English alphabet algorithms

Pointer Versions		Array Versions	
LFBM:	151.7	LFBM:	164.8
SS:	178.0	SS:	189.7
OMI1:	190.8	MSI1:	203.4
MSI1:	192.0	BM1F:	204.5
BM1F:	194.6	OMI1:	204.6
QSI1:	194.7	QSI1:	205.2
OMI2:	197.6	MSI2:	209.4
MSI2:	200.2	OMI2:	210.0
QSI2:	201.7	BM2F:	211.5
BM2F:	202.8	QSI2:	211.9
BM1S:	269.2	BM1S:	290.7
QS:	316.6	QS:	294.9
SLFC:	342.4	QS2:	381.3
BFI:	375.6	BM2S:	386.3
BM2S:	386.2	OM:	451.6
KMPI:	391.3	MS:	457.7
QS2:	415.9	SLFC:	469.0
OM:	456.8	BFI:	505.2
MS:	458.6	KMPI:	517.4
BF:	1014.5	BF:	1076.3
KMP:	1482.1	KMP:	1402.4

Table 4.3. Text accessed vs. total text passed in search

```
                    TEXT ACCESSES/TOTAL TEXT

************************************************************


The total number of words is:  24474

Plen Words msi1/2 om/ms omi1/2  qs    ss   lfbm bm1f/2 bm1s2
---- ----- ------ ----- ------  --    --   ---- ------ -----
 1    26    1.08   1.03  1.08  1.08  1.00  1.09  1.05   1.03
 2    91    0.73   0.71  0.74  0.76  0.56  0.56  0.57   0.72
 3   759    0.57   0.56  0.57  0.59  0.40  0.40  0.41   0.57
 4  2142    0.47   0.46  0.47  0.49  0.31  0.31  0.32   0.48
 5  3097    0.40   0.40  0.41  0.42  0.26  0.25  0.27   0.41
 6  3796    0.36   0.35  0.36  0.37  0.23  0.22  0.24   0.37
 7  4045    0.32   0.32  0.32  0.34  0.20  0.20  0.21   0.33
 8  3578    0.30   0.29  0.30  0.31  0.18  0.18  0.19   0.31
 9  2970    0.28   0.27  0.28  0.29  0.17  0.16  0.18   0.29
10  1890    0.26   0.26  0.26  0.28  0.16  0.15  0.17   0.27
11  1072    0.25   0.24  0.25  0.26  0.15  0.15  0.16   0.26
12   547    0.24   0.23  0.24  0.25  0.14  0.14  0.15   0.24
13   275    0.23   0.22  0.23  0.24  0.14  0.13  0.14   0.24
14   111    0.22   0.21  0.22  0.23  0.13  0.13  0.14   0.23
15    41    0.21   0.20  0.21  0.22  0.12  0.12  0.13   0.22
16    17    0.20   0.19  0.20  0.21  0.12  0.12  0.13   0.21
17     8    0.19   0.18  0.19  0.20  0.11  0.11  0.12   0.20
18     5    0.19   0.18  0.19  0.20  0.10  0.10  0.11   0.18
20     1    0.19   0.17  0.19  0.20  0.10  0.10  0.10   0.18
21     2    0.17   0.16  0.17  0.18  0.09  0.09  0.09   0.17
22     1    0.16   0.15  0.16  0.17  0.08  0.09  0.09   0.16

the number of mis_matches = 0
```

Table 4.3. Text accessed vs. total text (cont)

```
             TEXT ACCESSES/TOTAL TEXT

********************************************


The total number of words is:  24474

Plen Words  kmp    kmpi    bf    bfi    slfc
---- -----  ----   ----    --    ---    ----
 1    26    1.06   1.12   1.00   1.05   1.05
 2    91    1.04   1.04   1.04   1.04   1.03
 3   759    1.04   1.04   1.04   1.04   1.04
 4  2142    1.04   1.04   1.04   1.04   1.04
 5  3097    1.04   1.04   1.05   1.05   1.03
 6  3796    1.04   1.04   1.05   1.05   1.03
 7  4045    1.04   1.04   1.05   1.05   1.03
 8  3578    1.05   1.05   1.05   1.05   1.03
 9  2970    1.05   1.05   1.06   1.06   1.03
10  1890    1.05   1.05   1.06   1.06   1.03
11  1072    1.06   1.06   1.06   1.06   1.03
12   547    1.06   1.06   1.07   1.07   1.03
13   275    1.06   1.06   1.07   1.07   1.03
14   111    1.06   1.06   1.07   1.07   1.03
15    41    1.07   1.07   1.08   1.08   1.02
16    17    1.07   1.07   1.08   1.08   1.02
17     8    1.06   1.06   1.07   1.07   1.02
18     5    1.09   1.09   1.10   1.10   1.03
20     1    1.10   1.10   1.11   1.11   1.02
21     2    1.09   1.09   1.09   1.09   1.03
22     1    1.10   1.10   1.11   1.11   1.02

the number of mis_matches = 0
```

Table 4.4. Ishift loop algorithms vs. no ishift loop

NO ISHIFT		ISHIFT	
OM:	456.8	OMI1:	190.8
		OMI2:	197.6
MS:	458.6	MSI1:	192.0
		MSI2:	200.2
BM1S:	269.2	BM1F:	194.6
BM2S:	386.2	BM2F:	202.8
QS:	316.6	QSI1:	194.7
QS2:	415.9	QSI2:	201.7
BF:	1014.5	BFI:	375.6
KMP:	1482.1	KMPI:	391.3
		LFBM:	151.7
		SS:	178.0
		SLFC:	342.4

****

Table 4.5. Delta1 algorithms vs. delta1 and delta2

DELTA1		DELTA1,DELTA2	
OMI1:	190.8	OMI2:	197.6
MSI1:	192.0	MSI2:	200.2
BM1F:	194.6	BM2F:	202.8
QSI1:	194.7	QSI2:	201.7
BM1S:	269.2	BM2S:	386.2
QS:	316.6	QS2:	415.9
LFBM:	151.7		
SS:	178.0		
		OM:	456.8
		MS:	458.6

Table 4.6. Quaternary expanded alphabet results

POINTER VERSIONS: { A, C, G, T }

TIMES

plen	#words	ss-time	acgt-time	acgt2-time
50	500	1.3	0.6	0.7
100	500	3.0	0.9	1.0
150	500	3.9	0.9	1.0
200	500	5.5	1.0	1.2
450	500	12.6	1.7	2.1

****

ARRAY VERSIONS: { A, C, G, T }

TIMES

plen	#words	ss-time	acgt-time	acgt2-time
50	500	1.3	0.6	0.7
100	500	3.0	0.9	1.0
150	500	3.9	0.9	1.0
200	500	5.5	1.0	1.2
450	500	12.6	1.7	2.1

****

TEXT ACCESSED/TOTAL TEXT

plen	#words	ss-acc	acgt-acc	acgt2-acc
50	500	.53	.1	.1
100	500	.54	.06	.06
150	500	.53	.05	.05
200	500	.52	.04	.04
450	500	.53	.03	.03

****

Table 4.7. Binary expanded alphabet results

POINTER VERSIONS: { A, T }

TIMES

plen	#words	ss-time	acgt/2-time	bin/2-time
50	500	5.9	1.4 / 1.3	0.8 / 0.7
100	500	12.1	2.6 / 2.1	0.9 / 1.0
150	500	17.6	4.0 / 2.8	1.0 / 1.2
200	500	24.5	5.3 / 3.5	1.3 / 1.4
450	500	55.1	12.6 / 7.2	2.1 / 2.5

****

ARRAY VERSIONS: { A, T }

TIMES

plen	#words	ss-time	acgt/2-time	bin/2-time
50	500	5.0	1.5 / 1.3	0.8 / 0.8
100	500	9.6	2.7 / 2.1	0.9 / 1.0
150	500	14.4	4.0 / 2.8	1.0 / 1.2
200	500	23.0	5.6 / 3.6	1.2 / 1.4
450	500	50.2	14.1 / 8.1	2.5 / 3.0

****

TEXT ACCESSED/TOTAL TEXT

plen	#words	ss-acc	acgt/2-acc	bin/2-acc
50	500	1.92	.34 / .27	.20 / .19
100	500	1.98	.32 / .23	.11 / .11
150	500	1.92	.33 / .21	.08 / .08
200	500	1.98	.32 / .20	.07 / .07
450	500	1.97	.32 / .17	.05 / .05

****

Chapter Five

Discussion

5.1 English Alphabet Algorithms

The least-frequent-Boyer-Moore (lfbm) and str_search (ss) outperform all other algorithms tested. Lfbm performs the best in both timed runs and as well as the text accessed versus total text measure. Other strong algorithms include optimal mismatch and maximal shift with ishift loop (omi1, omi2, msi1, msi2), Boyer-Moore fast with ishift loop (bm1f, bm2f) and quick search with ishift loop (qsi1, qsi2).

Adding the ishift loop to any string searching algorithm dramatically improves performance. In most cases, algorithms with an ishift loop improve running times by a factor of 2. In the case of Knuth-Morris-Pratt, the algorithm with the ishift loop analogue runs more than three times faster than the "standard" KMP algorithm. Sunday's algorithms all run approximately 2.5 times faster with the ishift loop. Text accesses however, are not different between versions with the ishift loop and those without. When iterating in the ishift loop, the text string still must be accessed in order to lookup the ishift value in the delta1 table.

The tests conducted show that adding a delta2 table to any algorithm does not improve performance. In all cases, versions with a delta1 table outperform versions with both a delta1 and a delta2 table. This includes Sunday's algorithms since his TD1 and TD2 tables are based on the exact same heuristics as the Boyer-Moore delta tables.

Sunday's algorithms do not outperform the Boyer-Moore algorithm as he claims. In timed runs, optimal mismatch (om) and maximal shift (ms) run slower than all Boyer-Moore versions tested. Quick search runs faster than om and ms, however it still runs slower than all Boyer-Moore versions except bm2s.

Empirical results show that Sunday's algorithms access the text string more frequently than do any of the Boyer-Moore algorithms. In fact, the text accessed versus total text results of this paper differ substantially from the same runs made and published in Sunday's paper [8]. Om and ms access the text string about 2 percent less than bm1s and bm2s however, ss accesses the text string approximately 25 percent less than any of Sunday's algorithms. The bm1f and bm2f algorithms access the text string just slightly fewer times than Sunday's algorithms. With the addition of the ishift loop, Sunday's algorithms do exhibit improved performance however, that is the work of this paper and not Sunday's.

5.2 Expanded Alphabet Algorithms

Random binary and quaternary alphabet strings were generated to test the expanded alphabet algorithms. The ss algorithm was included in runs with these algorithms as a measure of comparison with a good performing English alphabet algorithm. Both the binary and quaternary algorithms were run on binary strings in order to determine how efficient the tailor made binary algorithms performed.

In the test runs performed, it is empirically shown that adding a delta2 table does not improve performance and in almost all cases it hinders performance. Any time saved by using a delta2 shift is apparently offset by the time it takes to build the delta2 array.

Both the binary (at) and the quaternary (acgt) algorithms display fast run times. They also access the text string fewer times than any algorithm tested. This is due to the parallel nature of the algorithm which enables it to make large consistent jumps in the text string while in the ishift loop. In comparison to ss, both expanded alphabet algorithms display excellent performance.

Chapter Six

Conclusion

This paper has attempted to examine pattern matching algorithms through empirical analysis. This method has shown that two long standing ideas of Boyer and Moore, the "fast loop" and the delta1 table, increase performance in any exact pattern matching algorithm. In every algorithm tested, analogues of the fast loop have dramatically decreased search times. It has been shown that the delta1 table when used without the delta2 table maximizes search efficiency in the tests conducted. In no case did an algorithm using both a delta1 and a delta2 table outperform an analogous version using just a delta1 table. Furthermore, considering the difficulty in understanding the delta2 shift, it is suggested the delta2 table be left out of any exact pattern matching algorithm used for searching English language text or nucleic acid sequences.

The least frequent Boyer Moore algorithm performs the best of all algorithms tested. The simplified version of the Boyer-Moore algorithm, str_search, also performs very well. It is a highly recommended algorithm due to its brevity, simplicity and efficiency.

It has been shown that Sunday's algorithms do not outperform the Boyer-Moore algorithm, contrary to his claim. With the addition of the ishift loop however, the optimal mismatch and maximal shift algorithms do give good performance.

Expanded alphabet algorithms for processing binary and quaternary strings perform considerably better than any of the other algorithms when tested on random strings.

Of all the algorithms presented in this paper, none stand out as a panacea for all pattern matching applications. It is important to examine the criteria under which specific string searching is being conducted and choose the algorithm which best suits those conditions.

Appendix A
C-Language Implementaion

Listing 1: Quaternary expanded alphabet function: acgt2.

This function searches strings composed of characters from a four character alphabet: { A, C, G, T }. It uses two delta tables and has its own version of the ishift loop. Also included are all pattern preprocessing functions.

```
int acgt2( char text[], int tlen, char pat[], int plen )

{    int   delta1[MAX_ALPHABET], delta2[MAX_PATTERN_LENGTH],
           delta3[MAX_ALPHABET], delta4[MAX_ALPHABET];

     int i, j, m, ishift, encoded_4;
     char *tx, *p;            /* string scan pointers */
     char *last_text_char = text + tlen;

     /* call pattern preprocessing functions */
     create_delta1_2(delta1, delta2, pat, plen);
     create_delta3(delta3, pat, plen);
     create_delta4(delta1, delta3, delta4);

     /* concat pattern onto end of text  */
     for ( i = tlen, j = 0; (i < MAX_TEXT_LENGTH) &&
                          (i < tlen + plen); i++, j++ )
          text[i] = pat[j];
     text[plen+tlen] = '\0';

     tx = text + plen - 1;
     while (1) {
          if (tx > last_text_char)  /* no match found */
               return -1;

          /* calculate initial encoding  */
          encoded_4 = 0;
          for (m = 0; m < 4; ++m) {
               encoded_4 <<= 2;
               switch ( *(tx - 3 + m) )  {
                    case 'a': break;
                    case 'c': encoded_4 |= 0x00000001;
                              break;
                    case  'g':  encoded_4   |=   0x00000002;
                              break;
                    case  't':  encoded_4   |=   0x00000003;
                              break;
               }
          }
          /* ishift loop */
          ishift = delta4[encoded_4];
          while ( ishift != 0 ) {
               tx += ishift;
               encoded_4 = 0;
               for (m = 0; m < 4; ++m) {
                    encoded_4 <<= 2;
                    switch ( *(tx - 3 + m) ) {
                        case 'a': break;
                        case 'c':  encoded_4  |=  0x00000001;
                                  break;
                        case 'g':  encoded_4  |=  0x00000002;
                                  break;
                        case 't':  encoded_4  |=  0x00000003;
                                  break;
                    }
               }
               ishift = delta4[encoded_4];
          }

          tx -= 4;            /* since ishift == 0 then */
          p = pat + plen - 5; /* last 4 chars match     */

          if ( tx < last_text_char )
               while (1) {
                    if (p < pat)  /* if yes; found match */
                         return (tx - text + 1);

                    if ( *tx == *p ) {
                         p--;
                         tx--;
                    }
                    else break;
               }

          /* false start; increment text string pointer   */
          tx += delta2[(p - pat)];
     }
}

/**********************************************************/


void create_delta1_2(int *delta1, int *delta2,
                                   char *pat, int plen)

{    int i, j, k;
     int t, tp, f[MAX_PATTERN_LENGTH];

     /* initialize delta1 array */
     for (i = 0; i < MAX_ALPHABET; ++i)   delta1[i] = plen;
     /* fill in delta1 array with values */
     for (i = plen - 4; i >= 0; i--) {
          j = create_4_hex(pat, i); /* call encoding func */
          if (delta1[j] == plen)
               delta1[j] = plen - i - 4;
     }

     /* initialize delta2 array */
     for (j = 0; j < plen; j++)
          delta2[j] = (2 * plen - j - 1);

     j = plen - 1;
     t = plen;

     while (j >= 0)  {
          f[j] = t;
          while ( t <= plen-1 && pat[j] != pat[t] )  {
               delta2[t] = ( delta2[t] < (plen-j-1) ) ?
                    delta2[t] : (plen-j-1);
               t = f[t];
          }
          j--;  t--;
     }

     for (k = 0; k <= t; k++)
          delta2[k] = ( delta2[k] < (plen + t - k) ) ?
                    delta2[k] : (plen + t - k);

/*  This next section of code installs the correct delta2
    values in case there is a highly repetitive pattern such
    as 'aaaaa'.  (G. De V. Smit, 1982).  */

     tp = f[t];
     while (t <= plen - 1)  {
          while (t <= tp)  {
               delta2[t] = ( delta2[t] < (tp - t + plen) ) ?
                    delta2[t] : (tp - t + plen);
               t++;
          }
          tp = f[tp];
     }
}

/**********************************************************/


void create_delta3( int *delta3, char *pat, int plen )

{    int encoded_3, k;

     for ( k = 0; k < MAX_ALPHABET; k++ )
          delta3[k] = plen;
     /* encode leftmost 3 characters of pattern */
     encoded_3 = 0;
     for ( k = 0; k < 3; k++ )  {
          encoded_3 <<= 2;
          switch (pat[k]) {
               case 'a': break;
               case 'c': encoded_3 |= 0x00000001; break;
               case 'g': encoded_3 |= 0x00000002; break;
               case 't': encoded_3 |= 0x00000003; break;
          }
     }


     /* 4 different values will receive plen-3 */
     for ( k = encoded_3; k < MAX_ALPHABET; k += 64 )
          delta3[k] = plen - 3;

     /* get encoding for leftmost 2 characters of pattern */
     /* by right shifting by 2                            */
     encoded_3 >>= 2;

     /* 16 delta1 values take plen-2 as their array entry */
     for ( k = encoded_3; k < MAX_ALPHABET; k += 16 )
          if ( delta3[k] == plen )
               delta3[k] = plen - 2;

     /* get encoding for leftmost 1 character of pattern  */
     encoded_3 >>= 2;
     for ( k = encoded_3; k < MAX_ALPHABET; k += 4 )
          if ( delta3[k] == plen )
               delta3[k] = plen - 1;
}

/**********************************************************/


void create_delta4( int *delta1, int *delta3, int *delta4 )

{    int i;

     /* get max( delta1[i], delta3[i] ) */
     /* install max value into delta4   */
     for (i = 0; i < MAX_ALPHABET; ++i)
          delta4[i] = (delta1[i] < delta3[i]) ?
                                   delta1[i] : delta3[i];
}

/**********************************************************/


int create_4_hex( char *seq, int position )
```

```
{    int i, upper_lim, encoded_4;
     encoded_4 = 0;
     upper_lim = 4;

     if (position < 0) {
          upper_lim = position + 3;
          position = 0;
     }

     /* encode pattern substring passed into function     */
     for (i = 0; i < upper_lim; ++i) {
          encoded_4 <<= 2;
          switch (seq[position + i]) {
               case 'a': break;
               case 'c': encoded_4 |= 0x00000001; break;
               case 'g': encoded_4 |= 0x00000002; break;
               case 't': encoded_4 |= 0x00000003; break;
          }
     }
     return (encoded_4);
}
```

Listing 2: Brute Force

This is the standard brute force algorithm implemented with pointers.

```
int bf( char text[], int tlen, char pat[], int plen )

{    char *tx = text;          /*  text pointer    */
     char *p = pat;            /*  pattern pointer */
     char *text_end_ptr = text + tlen;
     char *pat_end_ptr = pat + plen;

     while ( (p != pat_end_ptr) && (tx <= text_end_ptr) )  {
          if ( *tx == *p )  {
               tx++;  p++;
          }  else  {           /*    back up!      */
               tx = tx - (p - pat) + 1;
               p = pat;
          }
     }

     if ( p == pat_end_ptr )   /* match found      */
          return (tx - text - plen);
     else                      /* no match found   */
          return -1;
}
```

Listing 3: Brute Force with ishift loop analogue

This is the brute force algorithm implemented with pointers and an ishift loop analogue.

```
int bfi( char text[], int tlen, char pat[], int plen )

{    char *tx = text;         /*  text pointer    */
     char *p = pat;           /*  pattern pointer */
     char first_pat_char = pat[0];
     char *last_match_ptr = text + tlen - plen;
     char *text_end_ptr = text + tlen;
     char *pat_end_ptr = pat + plen;
     int i, j;

     /* concat pattern onto end of text */
     for ( i = tlen, j = 0; (i < MAX_TEXT_LENGTH) &&
                            (i < tlen + plen); i++, j++ )
          text[i] = pat[j];
     text[plen + tlen] = '\0';

     do { /* ishift loop analogue */
          while ( *tx != first_pat_char )
               tx++;

          if ( tx < text_end_ptr)  {
               tx++;
               p++;
          }

          if ( tx < text_end_ptr && *tx == *p )
               do { p++;
                    tx++;
               }  while ( *tx == *p && p < pat_end_ptr );

          if ( p < pat_end_ptr ) {   /* false start?    */
               tx -= p - pat - 1;
               p = pat;
          }

     }  while  ( tx <= last_match_ptr && p < pat_end_ptr );

     if ( p == pat_end_ptr )          /* match found     */
          return (tx - plen - text);
     else                             /* no match found  */
          return -1;
}
```

Listing 4: Boyer-Moore: bm2_fast.

This function uses two delta tables and also uses the 'fast' loop adopted from the Implementation section ot the Boyer-Moore paper [1]. Text accesses and total text counts are also included in this listing.

```
int bm2_fast (char text[], int tlen, char pat[], int plen,
                       int *total_text, int *text_accessed )

{    char *tx = text + plen - 1;   /* text string pointer */
     char *p;                   /* pattern string pointer */
     int temp_tx;
     int large_pattern_value = delta1[ *reset_p ];
     char *reset_p = pat + plen - 1;
     char *last_text_char = text + tlen;

     int delta1[ MAX_ALPHABET ];
     int delta2[ MAX_PATTERN_LENGTH ];

     /* call pattern preprocessing function */
     init_delta1_2( pat, plen, tlen, delta1, delta2 );

     if (plen > tlen)   /* error check input */
          return -1;

     while (1)  {
          /* the 'fast' loop */
          do  {
               tx += delta1[ *tx ];
                (*text_accessed)++;
          }  while ( tx <= last_text_char );

          /* if no text left, no match found */
          if ( (tx - text) < large_pattern_value )  {
               *total_text = tlen;
               return -1;
          }

          /* text still left, subtract off 'large' value */
          tx -= large_pattern_value;

          p = reset_p;          /* reset pattern pointer */

          /* start doing comparisons between characters  */
          while ( (*text_accessed)++ && (*tx == *p) ) {
               if (p == pat)  {         /* found a match */
                    *total_text = (tx - text + plen);
                    return (tx - text);
               }
               tx--;
               p--;
          }

          /* false start; increment text string pointer  */
          temp_tx = delta1[ *tx ];
          (*text_accessed)++;
          tx += ( (temp_tx > plen) ||
                    (temp_tx < delta2[ p - pat ]) ) ?
                                delta2[ p - pat ] : temp_tx;
     }
}
```

Listing 5: Knuth-Morris-Pratt

This is the standard Knuth-Morris-Pratt algorithm implemented with pointers. The function which builds the 'next' table is also included.

```
int kmp ( char text[], int tlen, char pat[], int plen )

{    char *tx = text;       /* text string pointer    */
     char *p  = pat;        /* pattern string pointer */
     char *beyond_text = text + tlen;
     char *beyond_pat  = pat + plen;
     int   next[ MAX_PATTERN_LENGTH ];
     int i;

     /* make next table */
     initpattern( pat, plen, next );

     do  {
          if ( p < pat )  {
               tx++;
               p++;
          }
          if ( *tx == *p ) {
               tx++;
               p++;
          }  else                       /* false start */
               p = pat + next[p - pat];
     }  while ( p < beyond_pat && tx < beyond_text );

     if ( p >= beyond_pat )           /* found a match */
          return ( tx - text - plen );
     else                            /* no match found */
          return -1;
}

/**********************************************************/
```

```
void initpattern( char pstr[], int plen, int next[] )

/* function which installs values into 'next' table to
   assist KMP algorithm.          */

{    int i, j;

     next[0] = -1;

     for( i = 1; i < plen; i++ ) {
          j = next[i-1];
          while( j >= 0 && pstr[j] != pstr[i-1] )
               j = next[j];
          next[i] = (j+1);
     }

     for( i = 1; i < plen; i++ ) {
          if ( next[i] == 0 )
               continue;
          if ( pstr[i] == pstr[next[i]] )
               next[i] = next[next[i]];
     }
}
```

Listing 6: Knuth-Morris-Pratt with ishift loop analogue

The KMP algorithm implemented with pointers and using the ishift loop analogue.

```
int kmpi ( char text[], int tlen, char pat[], int plen )

{    char *tx = text;           /* text string pointer    */
     char *p =  pat;            /* pattern string pointer */
     char *beyond_text = text + tlen;
     char *beyond_pat  = pat + plen;
     char  first_pat_char = pat[0];
     int   next[ MAX_ALPHABET ];
     int i;

     /* build 'next' table */
     initpattern( pat, plen, next );

     text[tlen] = first_pat_char;

     do { if ( p < pat ) {
               tx++;
               p++;
          }
          if ( p == pat )   {
               /* ishift loop analogue */
               while ( *tx != first_pat_char )
                    tx++;
               if ( tx < beyond_text )   {
                    tx++;  p++;
               }
          }

          /* do character to character comparisons */
          while ( *tx == *p && tx < beyond_text ) {
               tx++;
               p++;
          }

          if ( p < beyond_pat )   /* false start?  */
               p = pat + next[p - pat];

     }  while ( p < beyond_pat && tx < beyond_text );

     if ( p >= beyond_pat )      /* found a match */
          return ( tx - text - plen );
     else                       /* no match found */
          return -1;
}
```

Listing 7: Least frequent Boyer-Moore

This algorithm attempts to maximize the time spent in the ishift loop by comparing the least frequent character of the pattern first. If that character is a mis-match, the ishift loop is reentered. It uses only one delta table: delta1.

```
int lfbm ( char *text, int tlen, char *pat, int plen )

{    float temp_freq = 25.0;   /* dummy frequency value */
     int  i, j, k;
     int delta1[ MAX_ALPHABET ];
     char *reset_p = pat + plen - 1;
     char *last_text_char = text + tlen;
     char *tx = text + plen - 1;
     char *temp_tx;
     char *p = pat;
     char lfc;
     int ishift, from_pat_end;


     if ( plen > tlen )        /* error check input     */
          return -1;
     comp_delta( plen, pat, delta1 );

     /* concat pattern onto end of text */
     for ( i = tlen, j = 0; (i < MAX_TEXT_LENGTH) &&
                              (i < tlen + plen); i++, j++ )
          text[i] = pat[j];
     text[plen+tlen] = '\0';

     /* find least frequent pattern char */
     for ( j = 0; j < plen; j++ )
          if ( Freq[ pat[j] ] < temp_freq )   {
               temp_freq = Freq[ pat[j] ];
               k = j;
               lfc = pat[j];
          }

     from_pat_end = plen - k - 1;

     while ( tx < last_text_char )   {
          p = reset_p;

               do  {
                    do  {  /* ishift loop */
                         ishift = delta1[ *tx ];
                         tx += ishift;
                    }  while ( ishift != 0 );

                    if ( *(tx - from_pat_end) == lfc )   {
                         temp_tx = tx;
                         break; /* chars match; exit loop */
                    }

                    /* chars didn't match, incre text ptr */
                    tx += delta1[ *(++tx) ];
               }  while (1);

               if( tx >= last_text_char ) /* ran off text */
                    return -1;     /* then no match found */

               do   /* start making comparisons */
               while ( (--p >= pat) && ( *p == *(--tx) ) );

               if ( p < pat )
                    return (tx - text);  /* found a match */
               else  {   /* false start; incre text ptr   */
                    tx = temp_tx;
                    tx += delta1[ *(++tx) ];
               }
     }
}
```

Listing 8: Search algorithm for om and ms, [8]

This algorithm is almost an exact copy of Sunday's algorithm given in [8]. A few changes have been made to speed up the search speed.

```
int search( char *text, int Tlen, char *pstr, int Plen )

{    PAT *p;                   /* pattern scan pointer */
     char *tx = text;          /* text scan pointer    */
     int TD1[ MAX_ALPHABET ];
     int TD2[ MAX_PATTERN_LENGTH ];
     int temp_tx;
     char *text_end_ptr = text + Tlen - Plen;
     int i;

     /* call preprocessing functions */
     build_TD1( pstr, Plen, TD1 );
     order_pattern( pstr, Plen, optimal_pcmp, pattern );
     build_TD2( pstr, Plen, TD2, pattern );

     while ( tx <= text_end_ptr )  {

          /* scan the pattern */
          for (p = pattern; p->c; ++p)
               if ( p->c != *(tx + p->loc) )
                    break;

          if (p->c == 0)   /* pat end=> got match     */
               return (tx - text);

          /* no match, so shift to next text position */
          temp_tx = TD1[ *(tx+Plen) ];
          tx += (temp_tx > TD2[ p-pattern ]) ?
                                 temp_tx : TD2[ p-pattern ];
     }

     return (-1);                      /* no match found   */
}
```

Listing 9: Search algorithm for om and ms with ishift loop analogue.

This search algorithm is patterned after Sunday's search algorithm except it uses an analogue of the ishift loop in an attempt to minimize search time.

```
int omi_2( char *text, int Tlen, char *pstr, int Plen )

{    PAT *p;                    /* pattern structure    */
     char *tx = text;           /* text string pointer */
     int TD1[ MAX_ALPHABET ];
     int TD2[ MAX_PATTERN_LENGTH ];
     char *text_end_ptr = text + Tlen - Plen;
     int i, j, d1;

     /* call preprocessing functions */
     build_TD1( pstr, Plen, TD1 );
     order_pattern( pstr, Plen, optimal_pcmp2, pattern );
     build_TD2( pstr, Plen, TD2, pattern );

     /* concat pattern onto end of text */
     for ( i = Tlen, j = 0; (i < MAX_TEXT_LENGTH) &&
                            (i < Tlen + Plen); i++, j++ )
          text[i] = pstr[j];
     text[Plen+Tlen] = '\0';

     while ( tx <= text_end_ptr )  {

          /* ishift loop analogue */
          while ( *tx != *pstr )
               tx += TD1[ *(tx + Plen) ];

          /* check for a match     */
          for (p = pattern; p->c; ++p)
               if ( p->c != *(tx + p->loc) )
                    break;

          if ( (p->c == 0)  && (tx <= text_end_ptr) )
               return (tx - text);   /* match found */

          /* false start; increment text pointer     */
          d1 = TD1[ *(tx+Plen) ];
          tx += ( d1 > TD2[ p-pattern ] ) ?
                            d1 : TD2[ p-pattern ];
     }

     return (-1);                    /* no match found */
}
```

Listing 10: Quick search

This is Sunday's quick search function. It uses only one delta table but is easily modified to take on two delta tables. The following is close to an exact duplicate to that given in [8].

```
int qs( char *text, int Tlen, char *pstr, int Plen )

{    char *p;                      /* pattern string pntr  */
     char *t, *tx = text;          /* text string pointers */
     int TD1[ MAX_ALPHABET ];
     char *text_end_ptr;

     build_TD1( pstr, Plen, TD1 );
     text_end_ptr = text + Tlen - Plen;

     while (tx <= text_end_ptr)  {

          /* scan pattern string */
          for (p = pstr, t = tx; *p; ++p, ++t)
               if ( *p != *t )
                    break;      /* mismatch, so stop     */

          if (*p == 0)
               return (tx - text);

          /* no substring match, so shift to next tx pos  */
          tx += TD1[ *(tx + Plen) ];  /* shift by delta1  */
     }

     return (-1);                    /* no substring found */
}
```

Listing 11: Sunday's preprocessing functions; called from qs, om and ms functions.

This listing contains all functions used by Sunday's search functions to preprocess the pattern. Included are the functions to build the delta1 and delta2 tables. Most of the code is identical to that given in [8].

```
void build_TD1(char *pstr, int Plen, int TD1[])

{    int i;
     char *p;

     /* initialize the TD1 table */
     for (i = 0; i < MAX_ALPHABET; i++)
          TD1[i] = Plen + 1;

     /* fill in the values from the pattern string */
     for (p = pstr; *p; p++)
          TD1[*p] = Plen - (p - pstr);

}

/*****************************************************/


void build_TD2(char *pstr, int Plen, int TD2[],
                                         PAT *pattern)

{    int lshift;
     int i, ploc;

     /* first init TD2[] for minimum matching left shift */
     TD2[0] = lshift = 1;    /* no preceeding chars, = 1 */

     for (ploc = 1; ploc < Plen; ++ploc)  {
          /* scan leftward for first matching shift */
          lshift = matchshift(pstr, Plen, pattern, ploc,
                                                   lshift);
          TD2[ploc] = lshift;  /* set initial match shift */
     }

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

/*****************************************************/


int matchshift( char *pstr, int Plen, PAT *pattern,
                                   int ploc, int lshift )

{    PAT *pat;
```

```
     int j;

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

Listing 12: Scan least frequent character

This algorithm checks whether the least frequent character of the pattern matches its aligned text character first. It uses an ishift analogue and does no preprocessing of the pattern.

```
int slfc ( char *text, int tlen, char *pat, int plen )

{    float temp_freq = 25.0;
     int  i, j, k;
     char ch;
     char *tx, *temp_tx, *p;

     if ( plen > tlen )
          return -1;

     for ( j = 0; j < plen; j++ )  /* get lfc of pattern */
          if ( Freq[ pat[j] ] < temp_freq )   {
               temp_freq = Freq[ pat[j] ];
               ch = pat[j];
               k = j;
          }

     tx = text + k;
     text[tlen] = ch;   /* concat lfc onto end of text  */

     do  {
          while ( *tx != ch  )  /* ishift loop analogue */
               tx++;

          if ( tx >= text + tlen )  /* pattern not found*/
```

```
               return -1;

          p = pat;
          temp_tx = tx;
          while ( (p - pat < plen) && *(tx - k) == *p )  {
               tx++;
               p++;
          }

          if ( p - pat == plen )     /* found a match    */
               return ( (tx - text) - k - plen);


          tx = temp_tx + 1;         /* resume search    */
     }  while (1);
}
```

Listing 13: str_search

This is the simplified Boyer-Moore algorithm which uses the ishift loop and 1 delta table.

```
int str_search( char text[], int Tlen, char pat[], int Plen)

{    int i, j, ishift;
     char *tx = text + Plen - 1;   /* text scan ptr    */
     char *p;                  /* pattern scan pointer */
     int delta[ MAX_ALPHABET ];
     int temp_tx;
     char *reset_p = pstr + Plen - 1;
     char *last_text_char = text + Tlen;

     /* get delta1 */
     compute_delta( Plen, pstr, delta );

     /* concat pattern onto end of text string */
     for ( i = Tlen, j = 0; (i < MAX_TEXT_LENGTH) &&
                            (i < Tlen + Plen); i++, j++ )
          text[i] = pstr[j];
     text[Plen+Tlen] = '\0';

     while ( tx < last_text_char )  {
          ishift = delta[ *tx ];

          /* the ishift loop */
          while ( ishift != 0 )  {
               tx += ishift;
               ishift = delta[ *tx ];
          }
          if( tx < last_text_char )  {
               p = reset_p;

               do    /* do pattern-text char comparisons */
               while ( (--p >= pstr) && ( *p == *(--tx) ) );

               if( p < pstr )
                    return tx - text;   /* found a match */
               else  {                  /* false start   */
                    temp_tx = delta[ *tx ];
                    tx += ( temp_tx > pstr + Plen - p ) ?
                                temp_tx : pstr + Plen - p;
               }

          } /* if */
     } /* while */

     return (-1);   /* no match found */
}
```

Appendix B
English Text Alphabet Frequencies

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

BIBLIOGRAPHY

1. Boyer, R. S., and Moore, J. S. A fast string searching algorithm. Communications ACM 20, 10 (Oct. 1977), 762-772.

2. Horspool, R. N. Practical fast searching in strings. Software-Practices and Experience, 10, 5 (May 1980), 501-506.

3. Knuth, D. E., Morris, J. H., and Pratt, V. R. Fast pattern matching in strings. SIAM J. Computing 6, 2 (June 1977), 323-350.

4. Park, S. K., and Miller, K. W. Random number generators: good ones are hard to find. Communications ACM, 31, 10, (Oct. 1988), 1192-1201.

5. Rabin, M. O., and Karp, R. M. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, 31, (March 1987), 249-260.

6. Sedgewick R., Algorithms, Addison-Wesley, Reading, MA, 1983.

7. Smit, G. V., A comparison of three string matching algorithms. Software-Practices and Experience, 12, 1 (Jan. 1982), 57-66.

8. Sunday, D. M., A very fast substring search algorithm. Communications ACM 33, 8 (Aug. 1990), 132-142.