A Fast Generic Sequence Matching Algorithm

David R. Musser  Gor V. Nishanov

Computer Science Department
Rensselaer Polytechnic Institute, Troy, NY 12180
{musser,gorik}@cs.rpi.edu
November 16, 2001

Abstract

A string matching—and more generally, sequence matching—algorithm is presented that has a linear worst-case computing time bound, a low worst-case bound on the number of comparisons (2n), and sublinear average-case behavior that is better than that of the fastest versions of the Boyer-Moore algorithm. The algorithm retains its efficiency advantages in a wide variety of sequence matching problems of practical interest, including traditional string matching; large-alphabet problems (as in Unicode strings); and small-alphabet, long-pattern problems (as in DNA searches). Since it is expressed as a generic algorithm for searching in sequences over an arbitrary type $T$, it is well suited for use in generic software libraries such as the C++ Standard Template Library. The algorithm was obtained by adding to the Knuth-Morris-Pratt algorithm one of the pattern-shifting techniques from the Boyer-Moore algorithm, with provision for use of hashing in this technique. In situations in which a hash function or random access to the sequences is not available, the algorithm falls back to an optimized version of the Knuth-Morris-Pratt algorithm.

KEY WORDS  String search  String matching  Pattern matching  Sequence matching  Generic algorithms  Knuth-Morris-Pratt algorithm  Boyer-Moore algorithm  DNA pattern matching  C++ Standard Template Library  STL  Ada  Literate programming
# Contents

1 Introduction .......................................................... 1

2 Linear and Accelerated Linear Algorithms ...................... 2

3 Benchmarking with English Texts .................................. 9

4 Hashed Accelerated Linear Algorithm .............................. 9

5 Searching for DNA Patterns ......................................... 13

6 Large Alphabet Case ................................................ 15

7 Generic Search Algorithms ......................................... 15

8 How to Obtain the Appendices and Code ......................... 23

9 Conclusion ............................................................. 24

A Tests of Expository Versions of the Algorithms ............... 27
   A.1 Algorithm Declarations ...................................... 27
   A.2 Simple Tests .................................................. 31
   A.3 Large Tests .................................................. 34
   A.4 Timed Tests .................................................. 36

B C++ Library Versions and Test Programs ...................... 38
   B.1 Generic Library Interfaces ................................... 39
      B.1.1 Library Files ............................................. 39
      B.1.2 Search Traits ............................................ 39
      B.1.3 Search Functions ....................................... 40
      B.1.4 Skip Table Computation .................................. 43
      B.1.5 Next Table Procedure and Call ......................... 43
   B.2 Experimental Version for Large Alphabet Case .......... 44
   B.3 DNA Search Functions and Traits ............................ 47
   B.4 Simple Tests ................................................ 48
   B.5 Large Tests ................................................ 52
   B.6 Timed Tests ................................................ 55
   B.7 Timed Tests (Large Alphabet) ............................... 57
   B.8 Counted Tests .............................................. 59
   B.9 Application to Matching Sequences of Words ............. 63
      B.9.1 Large Tests .............................................. 63
      B.9.2 Timed Tests .............................................. 65

C Index of Part Names ................................................. 66
1 Introduction

The traditional string matching problem is to find an occurrence of a pattern (a string) in a text (another string), or to decide that none exists. Two of the best known algorithms for the problem of string matching are the Knuth-Morris-Pratt [KMP77] and Boyer-Moore [BM77] algorithms (for short, we will refer to these as KMP and BM). Although KMP has a low worst-case bound on number of comparisons ($2n$, where $n$ is the length of the text), it is often considered impractical, since the number of comparisons it performs in the average case is not significantly smaller than that of the straightforward (SF) algorithm [Sm82], and the overhead for initialization is higher. On the other hand, despite the fact that BM has a higher worst-case bound on the number of comparisons ($\approx 3n$ [Cole96]), it has excellent sublinear behavior in the average case. This fact often makes BM the algorithm of choice in practical applications.

In [BM77], Boyer and Moore described both a basic version of their algorithm and an optimized version based on use of a “skip loop.” We will refer to the latter algorithm as Accelerated Boyer-Moore, or ABM for short. Unfortunately, this version remained unnoticed by most researchers despite its much better performance. For example, ABM outperforms the Quick Search [Su90] and Boyer-Moore-Horspool [Horspool88] improvements of the basic BM algorithm. This state of affairs was highlighted by Hume and Sunday in 1991 in [HS91], in which they introduced two algorithms, LC (least cost) and TBM (Tuned BM) [HS91], that perform faster than ABM in the average case. These two algorithms use the skip loop of ABM combined with variants of the straightforward algorithm that use information on character frequency distribution in the target text. For traditional string matching LC and TBM have excellent average case behavior, but in the worst case they behave like SF, taking time proportional to the product of the text and pattern lengths.

Even in the average case, the skip loop as it is used in ABM and other algorithms performs poorly with small alphabets and long patterns, as happens, for example, in the problem of DNA pattern matching. And if the alphabet is large, as for example with Unicode strings, initialization overhead and memory requirements for the skip loop weigh against its use.

This article describes a new linear string-matching algorithm and its generalization to searching in sequences over an arbitrary type $T$. The algorithm is based on KMP and has the same low $2n$ worst-case bound on number of comparisons, but it is better than ABM (comparable with TBM) in average case performance on English text strings. It employs a hash-coded form of the skip loop making it suitable even for cases with large alphabets or with small alphabets and long patterns. Since it is expressed as a generic algorithm for searching in sequences over an arbitrary type $T$, the new algorithm is well suited for use in generic software libraries such as the C++ Standard Template Library (STL). We present the algorithm in the following sections by starting with the basic KMP algorithm and transforming it with optimizations and addition of the skip loop in several alternative forms. The optimized form of KMP without the skip loop also
serves well in cases in which access to the sequences is restricted to forward, one-element-at-a-time iteration rather than random-access. We also discuss experimental results and some of the issues in including the new algorithm in a generic algorithm library.

2 Linear and Accelerated Linear Algorithms

Let \( m \geq a \geq 0 \) and \( n \geq b \geq 0 \) and suppose \( p_a \ldots p_{m-1} \) is a pattern of size \( m-a \) to be searched for in the text \( t_b \ldots t_{n-1} \) of size \( n-b \). Characters of the pattern and the text are drawn from an alphabet \( \Sigma \). The Knuth-Morris-Pratt algorithm can be viewed as an extension of the straightforward search algorithm. It starts comparing symbols of the pattern and the text from left to the right. However, when a mismatch occurs, instead of shifting the pattern by one symbol and repeating matching from the beginning of the pattern, KMP shifts the pattern to the right in such a way that the scan can be restarted at the point of mismatch in the text. The amount of shift is determined by precomputed function \( \text{next} \), defined by

\[
\text{next}(j) = \max_{i<j} \{i|p_a \ldots p_{i-1} = p_{a+j-i} \ldots p_{j-1} \land p_i \neq p_j\}
\]

(We let \( \text{next}(j) = a - 1 \) if there is no \( i \) satisfying the conditions.) Here is the basic KMP algorithm as it appeared in [KMP77], except that we use more general index ranges:\footnote{Although most authors use pseudocode for expository purposes, we prefer to be able to check all code with a compiler. The expository versions of algorithms in this paper are expressed in Ada 95, which has a syntax similar to that of most pseudocode languages (at least if one omits the details of subprogram headers and package declarations, which we include only in an appendix that deals with actual compilation of the code). The generic library components developed later in the paper are written in C++. Throughout the paper we present expository and production code in a variant of Knuth’s literate programming style [Knuth84], in which code is presented in “parts” numbered according to the page number on which they appear (with parts on the same page distinguished by appending a letter to the number). This form of presentation is supported by Briggs’ Nuweb tool [Briggs] (slightly modified, as discussed in a later section), with which we also generate all code files directly from the paper’s source file.}

(Basic KMP 2) ≡

\[
\begin{align*}
\text{pattern} &:= m - a; \ j := a; \ k := b; \\
\text{while} \ j < m \ \text{and} \ k < n \ \text{loop} \\
\text{while} \ j \geq a \ \text{and} \ \text{then} \text{text}(k) /= \text{pattern}(j) \ \text{loop} \\
\text{\quad} j := \text{next}(j); \\
\text{\quad} \text{end loop;}
\end{align*}
\]

\[
\begin{align*}
\text{\quad} k := k + 1; \ j := j + 1; \\
\text{end loop;}
\end{align*}
\]

\[
\begin{align*}
\text{if} \ j = m \ \text{then} \\
\text{\quad} \text{return} \ k - \text{pattern} - \text{size}; \\
\text{else} \\
\text{\quad} \text{return} \ n; \\
\text{end if;}
\end{align*}
\]

Used in part 27c.
A return value $i$ between $b$ and $n - \text{pattern.size}$ indicates a match found beginning at position $i$, while a return value of $n$ means there was no match. Although elegantly short, this algorithm does redundant operations along the expected execution path. That is, $\text{text}(k)$ is usually not equal to $\text{pattern}(j)$ and $\text{next}(j)$ is usually $a - 1$, so the inner loop usually sets $j$ to $a - 1$, redundantly tests it against $a$, and terminates. $k$ and $j$ are then both incremented and tested against their bounds, then $j$ is again redundantly compared with $a$. Knuth, Morris, and Pratt discussed a set of optimizations to the basic algorithm that required extending the text and pattern with additional characters, which is possible only under extra assumptions about the way the inputs are stored. We must avoid such assumptions when the goal is a generic algorithm. Instead, we eliminate the redundant operations by rewriting the algorithm in the following form, which we will call Algorithm L (for Linear) in this paper:

```
⟨Algorithm L, optimized linear pattern search 3a⟩ ≡
    pattern_size := m - a; k := b;
    ⟨Handle pattern size = 1 as a special case 3b⟩
    while k <= n - pattern_size loop
        ⟨Scan the text for a possible match 3c⟩
        ⟨Verify whether a match is possible at the position found 4a⟩
        ⟨Recover from a mismatch using the next table 4b⟩
        end loop;
    return n;
```

Used in part 27c.

The following code allows us to eliminate a test in the main loop:

```
⟨Handle pattern size = 1 as a special case 3b⟩ ≡
    if pattern_size = 1 then
        while k /= n and then text(k) /= pattern(a) loop
            k := k + 1;
        end loop;
        return k;
    end if;
```

Used in parts 3a, 5c, 7b, 12b, 27c.

The three parts of the body of the main loop are defined as follows:

```
⟨Scan the text for a possible match 3c⟩ ≡
    while text(k) /= pattern(a) loop
        k := k + 1;
        if k > n - pattern_size then
            return n;
        end if;
    end loop;
```

Used in parts 3a, 27c.
(Verify whether a match is possible at the position found 4a) ≡

\[
j := a + 1; k := k + 1;\\text{while text}(k) = \text{pattern}(j) \text{ loop}\\k := k + 1; j := j + 1;\\\text{if } j = m \text{ then}\\\quad \text{return } k - \text{pattern}\_\text{size};\\\text{end if;}\\\text{end loop;}
\]

Used in parts 3a, 27c.

(Recover from a mismatch using the next table 4b) ≡

\[
\text{loop}\\j := \text{next}(j);\\\text{if } j < a \text{ then}\\\quad k := k + 1; \text{exit;}\\\text{end if;}\\\text{exit when } j = a;\\\text{while text}(k) = \text{pattern}(j) \text{ loop}\\k := k + 1; j := j + 1;\\\text{if } j = m \text{ then}\\\quad \text{return } k - \text{pattern}\_\text{size};\\\text{end if;}\\\text{if } k = n \text{ then}\\\quad \text{return } n;\\\text{end if;}\\\text{end loop;}
\]

Used in parts 3a, 5c.

This last part guarantees linear worst-case behavior. Notice that if we simply replace the last part with the code 
\[
k := k - (j - a) + 1
\]
we obtain (an optimized form of) the straightforward algorithm.

Algorithm L can be further improved by incorporating a skip loop similar to the one that accounts for the excellent sublinear average time behavior of ABM. The idea of this technique is demonstrated in the following pair of examples:

Text: ......uuuuuuuuuuua.... ......uuuuuuuuuuue....

Before Shift: bcdabcdabcd bcdabcdabcd

After Shift: bcdabcdabcd bcdabcdabcd

We inspect the text character \(t_j\) that corresponds to the last character of the pattern, and if \(t_j \neq p_{m-1}\) we shift the pattern by the amount determined by the \text{skip} function, which maps any character of the alphabet to the range \([0, m-1]\) and is defined as follows:

\[
\text{skip}(x) = \begin{cases} 
m - a & \text{if } \forall j : a \leq j < m \Rightarrow p_j \neq x \\
m - 1 - i & \text{otherwise, where } i = \max\{j : a \leq j < m \land p_j = x\}
\end{cases}
\]
This is the same function as Boyer and Moore’s $\delta_1$ [BM77]. The following code replaces the scan part of Algorithm L:

(Scan the text using the skip loop 5a) ≡

```
loop
  d := skip(text(k + pattern_size - 1));
  exit when d = 0;
  k := k + d;
  if k > n - pattern_size then
    return n;
  end if;
end loop;
```

Used in part 5c.

If the exit is taken from this loop then $text(k + pattern_size - 1) = pattern(m - 1)$. We also change the verifying part of Algorithm L to the following:

(Verify the match for positions a through m - 2 5b) ≡

```
j := a;
while text(k) = pattern(j) loop
  k := k + 1; j := j + 1;
  if j = m - 1 then
    return k - pattern_size + 1;
  end if;
end loop;
```

Used in part 5c.

The algorithm incorporating these changes will be called the Accelerated Linear algorithm, or AL for short. In preliminary form, the algorithm is as follows:

(Accelerated Linear algorithm, preliminary version 5c) ≡

```
pattern_size := m - a; k := b;
  (Handle pattern size = 1 as a special case 3b)
  (Compute next table 30b)
  (Compute skip table and mismatch shift 6a)
while k <= n - pattern_size loop
  (Scan the text using the skip loop 5a)
  (Verify the match for positions a through m - 2 5b)
  if mismatch_shift > j - a then
    k := k + (mismatch_shift - (j - a));
  else
    (Recover from a mismatch using the next table 4b)
  end if;
end loop;
return n;
```

Used in part 29b.
Following the verification part, we know that the last character of the pattern and corresponding character of the text are equal, so we can choose whether to proceed to the recovery part that uses the next table or to shift the pattern by the amount mismatch_shift, predefined as

$$\text{mismatch\_shift} = \begin{cases} m - a & \text{if } \forall j : a \leq j < m - 1 \Rightarrow p_j \neq p_{m-1} \\ m - 1 - i & \text{otherwise, where } i = \max\{j : a \leq j < m - 1 \land p_j = p_{m-1}\} \end{cases}$$

This value can be most easily computed if it is done during the computation of the skip table:

\begin{verbatim}
(Compute skip table and mismatch shift 6a) ≡
for i in Character'Range loop
    skip(i) := pattern_size;
end loop;
for j in a .. m - 2 loop
    skip(pattern(j)) := m - 1 - j;
end loop;
mismatch_shift := skip(pattern(m - 1));
skip(pattern(m - 1)) := 0;
\end{verbatim}

Used in parts 5c, 7b.

The skip loop as described above performs two tests for exit during each iteration. As suggested in [BM77], we can eliminate one of the tests by initializing skip(pattern(m - 1)) to some value large, chosen large enough to force an exit based on the size of the index. Upon exit, we can then perform another test to distinguish whether a match of a text character with the last pattern character was found or the pattern was shifted off the end of the text string. We also add pattern_size - 1 to k outside the loop and precompute adjustment = large + pattern_size - 1.

\begin{verbatim}
(Scan the text using a single-test skip loop 6b) ≡
loop
    k := k + skip(text(k));
    exit when k >= n;
end loop;
if k < n + pattern_size then
    return n;
end if;
k := k - adjustment;
\end{verbatim}

Not used.

We can further optimize the skip loop by translating k by n (by writing k := k - n before the main loop), which allows the exit test to be written as k >= 0.
This saves an instruction over testing $k \geq n$, and a good compiler will compile $text(n + k)$ with only one instruction in the loop since the computation of $text + n$ can be moved outside. (In the C++ version we make sure of this optimization by putting it in the source code.) With this form of the skip loop, some compilers are able to translate it into only three instructions.

How large is $large$? At the top of the loop we have

$$k \geq b - n + pattern\_size - 1.$$ 

In the case in which $k$ is incremented by $large$, we must have

$$large + b - n + pattern\_size - 1 \geq pattern\_size.$$ 

Hence it suffices to choose $large = n - b + 1$.

We can also optimize the verification of a match by handling as a special case the frequently occurring case in which the first characters do not match.
(Verify match or recover from mismatch 8a) ≡

\[
\text{if text}(n + k) \neq \text{pattern}(a) \text{ then }
\]
\[
k := k + \text{mismatch_shift};
\]

else

(Verify the match for positions \(a + 1\) through \(m - 1\), with \(k\) translated 8b)

\[
\text{if \(\text{mismatch_shift} \geq j - a\) then}
\]
\[
k := k + (\text{mismatch_shift} - (j - a));
\]

else

(Recover from a mismatch using the next table, with \(k\) translated 8c)

end if;

end if;

end if;

Used in parts 7b, 12b.

The verification loop used here doesn’t really need to check position \(m - 1\), but we write it that way in preparation for the hashed version to be described later.

(Verify the match for positions \(a + 1\) through \(m - 1\), with \(k\) translated 8b) ≡

\[
j := a + 1;
\]

loop

\[
k := k + 1;
\]

exit when \text{text}(n + k) \neq \text{pattern}(j);

\[
j := j + 1;
\]

if \(j = m\) then

\[
\text{return } n + k - \text{pattern_size} + 1;
\]

end if;

end loop;

Used in part 8a.

(Recover from a mismatch using the next table, with \(k\) translated 8c) ≡

\[
\text{loop}
\]

\[
j := \text{next}(j);
\]

\[
\text{if } j < a \text{ then}
\]
\[
k := k + 1;
\]

exit;

end if;

exit when \(j = a\);

while \text{text}(n + k) = \text{pattern}(j) \text{ loop}

\[
k := k + 1; j := j + 1;
\]

if \(j = m\) then

\[
\text{return } n + k - \text{pattern_size};
\]

end if;

if \(k = 0\) then

\[
\text{return } n;
\]

end if;

end loop;

end loop;

Used in part 8a.
The AL algorithm thus obtained retains the same $2n$ upper case bound on the number of comparisons as the original KMP algorithm and acquires sublinear average time behavior equal or superior to ABM.

3 Benchmarking with English Texts

Before generalizing AL by introducing a hash function, let us consider its use as-is for traditional string matching. We benchmarked five algorithms with English text searches: a C++ version of SF used in the Hewlett-Packard STL implementation; L and AL in their C++ versions as given later in the paper and appendices; and the C versions of ABM [BM77] and TBM as given by Hume and Sunday [HS91]. (The version of AL actually used is the hashed version, HAL, discussed in the next section, but using the identity function as the hash function.)

We searched for patterns of size ranging from 2 to 18 in Lewis Carroll’s *Through the Looking Glass*. The text is composed of 171,556 characters, and the test set included up to 800 different patterns for each pattern size—400 text strings chosen at evenly spaced positions in the target text and up to 400 words chosen from the Unix spell-check dictionary (for longer pattern sizes there were fewer than 400 words). Table 1 shows search speeds of the five algorithms with code compiled and executed on three different systems:

1. g++ compiler, version 2.7.2.2, 60-Mh Pentium processor;
2. SGI CC compiler, version 7.10, SGI O² with MIPS R5000 2.1 processor;

These results show that HAL, ABM, and TBM are quite close in performance and are substantially better than the SF or L algorithms. On System 1, TBM is a slightly faster than HAL on the longer strings, but not enough to outweigh two significant drawbacks: first, like SF, it takes $\Omega(mn)$ time in the worst case; and, second, it achieves its slightly better average case performance though the use of character frequency distribution information that might need to be changed in applications of the algorithm other than English text searches. For both of these reasons, TBM is not a good candidate for inclusion in a library of generic algorithms.

For more machine independent performance measures, we show in a later section the number of operations per character searched, for various kinds of operations.

4 Hashed Accelerated Linear Algorithm

The skip loop produces a dramatic effect on the algorithm, when we search for a word or a phrase in an ordinary English text or in the text in some other natural or programming language with a mid-sized alphabet (26-256 characters, say). However, algorithms that use this technique are dependent on the alphabet size. In case of a large alphabet, the result is increased
<table>
<thead>
<tr>
<th>Pattern Size</th>
<th>Algorithm</th>
<th>System 1</th>
<th>System 2</th>
<th>System 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ABM</td>
<td>8.89665</td>
<td>24.6946</td>
<td>32.9261</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>8.26117</td>
<td>24.6946</td>
<td>32.9261</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>6.08718</td>
<td>24.6946</td>
<td>32.9261</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.28357</td>
<td>9.87784</td>
<td>24.6946</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td><strong>10.5142</strong></td>
<td><strong>32.9261</strong></td>
<td><strong>32.9261</strong></td>
</tr>
<tr>
<td>4</td>
<td>ABM</td>
<td>20.4425</td>
<td>46.7838</td>
<td>68.9446</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td><strong>23.3995</strong></td>
<td><strong>51.0369</strong></td>
<td><strong>83.6137</strong></td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>6.52724</td>
<td>27.8712</td>
<td>38.9093</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.29622</td>
<td>9.84923</td>
<td>23.3919</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>21.2602</td>
<td>49.123</td>
<td>71.4517</td>
</tr>
<tr>
<td>6</td>
<td>ABM</td>
<td>28.1637</td>
<td>60.2832</td>
<td>89.4829</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td><strong>31.2569</strong></td>
<td><strong>63.6323</strong></td>
<td><strong>108.055</strong></td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>6.45279</td>
<td>27.4015</td>
<td>37.9265</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.28142</td>
<td>9.84005</td>
<td>22.1973</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>29.2294</td>
<td>62.249</td>
<td>93.8837</td>
</tr>
<tr>
<td>8</td>
<td>ABM</td>
<td>33.7463</td>
<td>69.2828</td>
<td>106.674</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td><strong>37.0999</strong></td>
<td><strong>73.0482</strong></td>
<td><strong>126.801</strong></td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>6.34086</td>
<td>26.6684</td>
<td>36.5241</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.23323</td>
<td>9.78229</td>
<td>22.0342</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>35.3437</td>
<td>72.2627</td>
<td>112.007</td>
</tr>
<tr>
<td>10</td>
<td>ABM</td>
<td>39.6329</td>
<td>76.2308</td>
<td>117.47</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td><strong>42.5986</strong></td>
<td><strong>80.5134</strong></td>
<td><strong>135.202</strong></td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>6.32525</td>
<td>26.6383</td>
<td>36.1904</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.22537</td>
<td>9.74924</td>
<td>21.9134</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>41.1973</td>
<td>78.7439</td>
<td>125.714</td>
</tr>
<tr>
<td>14</td>
<td>ABM</td>
<td>47.7986</td>
<td>89.1214</td>
<td>129.631</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td><strong>49.8997</strong></td>
<td><strong>92.9962</strong></td>
<td><strong>147.511</strong></td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>6.22037</td>
<td>25.9262</td>
<td>33.6837</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.189</td>
<td>9.72233</td>
<td>21.1774</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>49.3573</td>
<td>92.9962</td>
<td>142.594</td>
</tr>
<tr>
<td>18</td>
<td>ABM</td>
<td>50.1514</td>
<td>97.859</td>
<td>141.352</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>50.1514</td>
<td><strong>101.773</strong></td>
<td><strong>159.021</strong></td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>5.86185</td>
<td>24.7023</td>
<td>31.4115</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.05173</td>
<td>9.63763</td>
<td>21.0275</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td><strong>51.2912</strong></td>
<td><strong>97.859</strong></td>
<td><strong>149.667</strong></td>
</tr>
</tbody>
</table>

Table 1: Algorithm Speed (Characters Per Microsecond) in English Text Searches on Three Systems
storage requirements and overhead for initialization of the occurrence table. Secondary effects are also possible due to architectural reasons such as cache performance. Performance of the skip loop is also diminished in cases in which the pattern size is much greater than the size of the alphabet. A good example of this case is searching for DNA patterns, which could be relatively long, say 250 characters, whereas the alphabet contains only four characters. In this section we show how to generalize the skip loop to handle such adverse cases.

The key idea of the generalization is to apply a hash function to the current position in the text to obtain an argument for the skip function.

(Scan the text using a single-test skip loop with hashing 11) \(\equiv\)

\[
\text{loop} \\
\text{k := k + skip(hash(text, n + k));} \\
\text{exit when k >= 0;} \\
\text{end loop;} \\
\text{if k < pattern_size then} \\
\text{return n;} \\
\text{end if;} \\
\text{k := k - adjustment;}
\]

Used in part 12b.

We have seen that the skip loop works well when the cardinality of domain of the skip function is of moderate size, say \(\sigma = 256\), as it is in most conventional string searches. When used with sequences over a type \(T\) with large (even infinite) cardinality, hash can be chosen so that it maps \(T\) values to the range \([0, \sigma)\). Conversely, if the cardinality of \(T\) is smaller than \(\sigma\), we can use more than one element of the text sequence to compute the hash value in order to obtain \(\sigma\) distinct values. In the context in which the skip loop appears, we always have available at least \(\text{pattern\_size}\) elements; whenever \(\text{pattern\_size}\) is too small to yield \(\sigma\) different hash values, we can either make do with fewer values or resort to an algorithm that does not use a skip loop, such as Algorithm L. (The skip loop is not very effective for small pattern lengths anyway.)

Of course, the skip table itself and the mismatch shift value must be computed using the hash function. Let \(\text{suffix\_size}\) be the number of sequence elements used in computing the hash function, where \(1 \leq \text{suffix\_size} \leq \text{pattern\_size}\).
(Compute skip table and mismatch shift using the hash function 12a)

\[
\text{for } i \text{ in hash_range loop}
\begin{align*}
  & \text{skip}(i) := \text{pattern\_size} - \text{suffix\_size} + 1; \\
  & \text{end loop;}
\end{align*}

\[
\text{for } j \text{ in } a + \text{suffix\_size} - 1 .. m - 2 \text{ loop}
\begin{align*}
  & \text{skip}(\text{hash(pattern, } j)) := m - 1 - j; \\
  & \text{end loop;}
\end{align*}
\]

\[
\text{mismatch\_shift} := \text{skip}(\text{hash(pattern, } m - 1));
\]

\[
\text{skip}(\text{hash(pattern, } m - 1)) := 0;
\]

Used in part 12b.

The remainder of the computation can remain the same, so we have the following algorithm in which it is assumed that the hash function uses up to \text{suffix\_size} elements, where \(1 \leq \text{suffix\_size} \leq \text{pattern\_size}\).

(Hashed Accelerated Linear algorithm 12b)

\[
\begin{align*}
\text{pattern\_size} := m - a; \text{ text\_size} := n - b; \ k := b;
\end{align*}
\]

(Handle pattern size = 1 as a special case 3b)

(Compute next table 30b)

(Compute skip table and mismatch shift using the hash function 12a)

\[
\text{large} := \text{text\_size} + 1;
\]

\[
\text{skip}(\text{hash(pattern, } m - 1)) := \text{large};
\]

\[
\text{adjustment} := \text{large} + \text{pattern\_size} - 1;
\]

\[
\ k := k - n;
\]

\[
\text{loop}
\begin{align*}
  & k := k + \text{pattern\_size} - 1; \\
  & \text{exit when } k >= 0;
\end{align*}
\]

(Scan the text using a single-test skip loop with hashing 11)

(Verify match or recover from mismatch 8a)

\[
\text{end loop;}
\]

\[
\text{return } n;
\]

Used in part 29a.

This algorithm will be called HAL. Note that AL is itself a special case of HAL, obtained using

\[
\text{hash(text, } k) = \text{text}(k) \quad \text{and (hence) } \text{suffix\_size} = 1,
\]

By inlining \text{hash}, we can use HAL instead of AL with minimal performance penalty (none with a good compiler).

It is also noteworthy that in this application of hashing, a “bad” hash function causes no great harm, unlike the situation with associative table searching in which hashing methods usually have excellent average case performance (constant time) but with a bad hash function can degrade terribly to linear time. (Thus, in a table with thousands of elements, searching might take thousands of times longer than expected.) Here the worst that can happen—with, say, a hash function that maps every element to the same
value—is that a sublinear algorithm degrades to linearity. As a consequence, in choosing hash functions we can lean toward ease of computation rather than uniform distribution of the hash values.

There is, however, an essential requirement on the hash function that must be observed when performing sequence matching in terms of an equivalence relation $\equiv$ on sequence elements that is not an equality relation. In this case, we must require that equivalent values hash to the same value:

$$x \equiv y \implies \text{hash}(x) = \text{hash}(y)$$

for all $x, y \in T$. We discuss this requirement further in a later section on generic library versions of the algorithms.

5 Searching for DNA Patterns

As an important example of the use of the HAL algorithm, consider DNA searches, in which the alphabet has only four characters and patterns may be very long, say 250 characters. For this application we experimented with hash functions $h_{ck}$ that map a string of $c$ characters into the integer range $[0, k)$. We chose four such functions, $h_{2,64}$, $h_{3,512}$, $h_{4,256}$, and $h_{5,256}$, all of which add up the results of various shifts of the characters they inspect. For example,

$$h_{4,256}(t, k) = (t(k - 3) + 2^2t(k - 2) + 2^4t(k - 1) + 2^6t(k)) \mod 256.$$  

The algorithms that use these hash functions bear the names HAL2, HAL3, HAL4, and HAL5, respectively. The other contestants were SF, L, ABM and the Giancarlo-Boyer-Moore algorithm (GBM), which was described in [HS91] and was considered to be the fastest for DNA pattern matching.

We searched for patterns of size ranging from 20 to 200 in a text of DNA strings obtained from [DNAsource]. The text is composed of 997,642 characters, and the test set included up to 80 different patterns for each pattern size—40 strings chosen at evenly spaced positions in the target text and up to 40 patterns chosen from another file from [DNAsource] (for longer pattern sizes there were fewer than 40 patterns). Table 2 shows search speeds of the five algorithms with code compiled and executed on the same three systems as in the English text experiments (the systems described preceding Table 1).

We see that each of the versions of HAL is significantly faster than any of the other algorithms, and the speed advantage increases with longer patterns—for patterns of size 200, HAL5 is over 3.5 to 5.5 times faster than its closest competitor, GBM, depending on the system. It appears that HAL4 is slightly faster than HAL5 or HAL3, but further experiments with different hash functions might yield even better performance.

---

13

---

2 In the C++ coding of this computation we use shifts in place of multiplication and masking in place of division. The actual C++ versions are shown in an appendix.
<table>
<thead>
<tr>
<th>Pattern Size</th>
<th>Algorithm</th>
<th>System 1</th>
<th>System 2</th>
<th>System 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>ABM</td>
<td>16.0893</td>
<td>37.3422</td>
<td>55.6074</td>
</tr>
<tr>
<td></td>
<td>GBM</td>
<td>25.8827</td>
<td>62.3888</td>
<td>138.267</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>13.1493</td>
<td>32.5853</td>
<td>53.8514</td>
</tr>
<tr>
<td></td>
<td>HAL2</td>
<td>28.7208</td>
<td>67.3143</td>
<td>146.168</td>
</tr>
<tr>
<td></td>
<td>HAL3</td>
<td>22.8165</td>
<td>63.9486</td>
<td>131.177</td>
</tr>
<tr>
<td></td>
<td>HAL4</td>
<td>21.9008</td>
<td>58.135</td>
<td>113.686</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>4.33091</td>
<td>16.3971</td>
<td>18.9477</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>3.28732</td>
<td>8.31851</td>
<td>16.94</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>18.0395</td>
<td>42.6324</td>
<td>63.1591</td>
</tr>
<tr>
<td>50</td>
<td>ABM</td>
<td>19.5708</td>
<td>44.693</td>
<td>70.6633</td>
</tr>
<tr>
<td></td>
<td>GBM</td>
<td>33.6343</td>
<td>78.046</td>
<td>193.67</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>13.6876</td>
<td>33.736</td>
<td>60.8033</td>
</tr>
<tr>
<td></td>
<td>HAL2</td>
<td>49.5795</td>
<td>106.716</td>
<td>290.505</td>
</tr>
<tr>
<td></td>
<td>HAL3</td>
<td>43.4625</td>
<td>106.716</td>
<td>275.215</td>
</tr>
<tr>
<td></td>
<td>HAL4</td>
<td>42.632</td>
<td>96.8349</td>
<td>249.004</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>4.3519</td>
<td>16.6003</td>
<td>19.439</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>3.28906</td>
<td>8.31333</td>
<td>16.7599</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>24.0764</td>
<td>54.4696</td>
<td>87.1514</td>
</tr>
<tr>
<td>100</td>
<td>ABM</td>
<td>21.2655</td>
<td>49.6781</td>
<td>73.4371</td>
</tr>
<tr>
<td></td>
<td>GBM</td>
<td>36.6439</td>
<td>85.8841</td>
<td>220.311</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>12.946</td>
<td>32.0706</td>
<td>56.3018</td>
</tr>
<tr>
<td></td>
<td>HAL2</td>
<td>70.4997</td>
<td>163.457</td>
<td>389.782</td>
</tr>
<tr>
<td></td>
<td>HAL3</td>
<td>71.2744</td>
<td>187.673</td>
<td>460.651</td>
</tr>
<tr>
<td></td>
<td>HAL4</td>
<td>70.4997</td>
<td>168.905</td>
<td>460.651</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>4.24474</td>
<td>16.0862</td>
<td>19.1938</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>3.24623</td>
<td>8.23929</td>
<td>16.5054</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>27.8368</td>
<td>66.6732</td>
<td>105.566</td>
</tr>
<tr>
<td>150</td>
<td>ABM</td>
<td>24.2269</td>
<td>56.1366</td>
<td>86.666</td>
</tr>
<tr>
<td></td>
<td>GBM</td>
<td>37.6383</td>
<td>86.667</td>
<td>205.834</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>14.0205</td>
<td>34.5456</td>
<td>60.9879</td>
</tr>
<tr>
<td></td>
<td>HAL2</td>
<td>84.3097</td>
<td>197.601</td>
<td>548.891</td>
</tr>
<tr>
<td></td>
<td>HAL3</td>
<td>91.641</td>
<td>247.001</td>
<td>548.891</td>
</tr>
<tr>
<td></td>
<td>HAL4</td>
<td>90.3318</td>
<td>235.239</td>
<td>494.002</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>4.3395</td>
<td>16.3037</td>
<td>19.5258</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>3.28992</td>
<td>8.33056</td>
<td>17.0935</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>29.1393</td>
<td>72.6474</td>
<td>107.392</td>
</tr>
<tr>
<td>200</td>
<td>ABM</td>
<td>23.9786</td>
<td>55.3853</td>
<td>86.3636</td>
</tr>
<tr>
<td></td>
<td>GBM</td>
<td>37.0578</td>
<td>90.9902</td>
<td>212.31</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>13.3106</td>
<td>33.3036</td>
<td>57.9028</td>
</tr>
<tr>
<td></td>
<td>HAL2</td>
<td>89.3449</td>
<td>221.541</td>
<td>509.545</td>
</tr>
<tr>
<td></td>
<td>HAL3</td>
<td>103.527</td>
<td>283.081</td>
<td>636.931</td>
</tr>
<tr>
<td></td>
<td>HAL4</td>
<td>105.196</td>
<td>283.081</td>
<td>566.161</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>4.26565</td>
<td>16.2275</td>
<td>19.0841</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>3.25946</td>
<td>8.28528</td>
<td>16.8167</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>28.7321</td>
<td>73.8471</td>
<td>113.232</td>
</tr>
</tbody>
</table>

Table 2: Algorithm Speed (Characters Per Microsecond) in DNA Searches on Three Systems
6 Large Alphabet Case

Suppose the alphabet $\Sigma$ has $2^{16} = 65,536$ symbols, as in Unicode for example. To use the skip loop directly we must initialize a table with 65,536 entries. If we are only going to search, say, for a short pattern in a 10,000 character text, the initialization overhead dominates the rest of the computation.

One way to eliminate dependency on the alphabet size is to use a large zero-filled global table $\text{skip1}(x) = \text{skip}(x) - m$, so that the algorithm fills at most $m$ positions with pattern-dependent values at the beginning, performs a search, and then restores zeroes. This approach makes the algorithm non-reentrant and therefore not suitable for multi-threaded applications, but it seems worth investigating for single-threaded applications.

Another approach is to use HAL with, say,

$$H(t, k) = t(k) \mod 256$$

as the hash function. In order to compare these two approaches we implemented a non-hashed version of AL using $\text{skip1}$, called NHAL, and benchmarked it against HAL with $H$ as the hash function. The C++ code for NHAL is shown in an appendix.

We searched for patterns of size ranging from 2 to 18 in randomly generated texts of size 1,000,000 characters, with each character being an integer chosen with uniform distribution from 0 to 65,535. Patterns were chosen from the text at random positions. The test set included 500 different patterns for each pattern size. Table 3 summarizes the timings obtained using the same three systems as described preceding Table 1. We can see that HAL demonstrates significantly better performance than NHAL. On systems 1 and 2 the ratio of HAL’s speed to NHAL’s is much higher than on system 3, and we attribute this disparity to poor optimization abilities of the compilers we used on systems 1 and 2.

We conclude that the hashing technique presents a viable and efficient way to eliminate alphabet-size dependency of search algorithms that use the skip loop.

7 Generic Search Algorithms

As we have seen, the HAL algorithm retains its efficiency advantages in a wide variety of search problems of practical interest, including traditional string searching with small or large alphabets, and short or long patterns. These qualities make it a good candidate for abstraction to searching in sequences over an arbitrary type $T$, for inclusion in generic software libraries such as the C++ Standard Template Library (STL) [StepanovLee, MS96].

By some definitions of genericity, HAL is already a generic algorithm, since the hash function can be made a parameter and thus the algorithm can be adapted to work with any type $T$. In STL, however, another important issue bearing on the generality of operations on linear sequences is the kind of access to the sequences assumed—random access or something weaker,
<table>
<thead>
<tr>
<th>Pattern Size</th>
<th>Algorithm</th>
<th>System 1</th>
<th>System 2</th>
<th>System 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>HAL</td>
<td>10.4369</td>
<td>27.3645</td>
<td>39.5632</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>7.1197</td>
<td>28.5543</td>
<td>41.5664</td>
</tr>
<tr>
<td></td>
<td>NHAL</td>
<td>6.63479</td>
<td>12.3915</td>
<td>26.4818</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.48334</td>
<td>9.83157</td>
<td>23.125</td>
</tr>
<tr>
<td>4</td>
<td>HAL</td>
<td>18.7455</td>
<td>44.375</td>
<td>64.3873</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>7.125</td>
<td>28.3082</td>
<td>41.5665</td>
</tr>
<tr>
<td></td>
<td>NHAL</td>
<td>10.9539</td>
<td>21.0497</td>
<td>47.5906</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.48611</td>
<td>9.86112</td>
<td>22.8038</td>
</tr>
<tr>
<td>6</td>
<td>HAL</td>
<td>25.3206</td>
<td>54.9243</td>
<td>86.7225</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>7.1179</td>
<td>28.4091</td>
<td>41.7146</td>
</tr>
<tr>
<td></td>
<td>NHAL</td>
<td>14.2358</td>
<td>28.1663</td>
<td>63.3741</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.505</td>
<td>9.86663</td>
<td>23.0451</td>
</tr>
<tr>
<td>8</td>
<td>HAL</td>
<td>31.1354</td>
<td>67.1919</td>
<td>94.0686</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>7.12946</td>
<td>28.6296</td>
<td>41.676</td>
</tr>
<tr>
<td></td>
<td>NHAL</td>
<td>16.9606</td>
<td>33.5959</td>
<td>80.3025</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.49445</td>
<td>9.88709</td>
<td>22.7062</td>
</tr>
<tr>
<td>10</td>
<td>HAL</td>
<td>35.7717</td>
<td>72.4895</td>
<td>112.484</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>7.09913</td>
<td>28.3655</td>
<td>41.2915</td>
</tr>
<tr>
<td></td>
<td>NHAL</td>
<td>19.1634</td>
<td>38.3768</td>
<td>98.8494</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.49017</td>
<td>9.85507</td>
<td>22.8114</td>
</tr>
<tr>
<td>14</td>
<td>HAL</td>
<td>42.9195</td>
<td>78.1701</td>
<td>149.234</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>7.1132</td>
<td>28.5491</td>
<td>41.0393</td>
</tr>
<tr>
<td></td>
<td>NHAL</td>
<td>23.5262</td>
<td>47.5818</td>
<td>136.798</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.48911</td>
<td>9.80043</td>
<td>22.7996</td>
</tr>
<tr>
<td>18</td>
<td>HAL</td>
<td>47.51862</td>
<td>96.9324</td>
<td>173.458</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>7.144521</td>
<td>28.1684</td>
<td>41.1963</td>
</tr>
<tr>
<td></td>
<td>NHAL</td>
<td>26.4274</td>
<td>56.8225</td>
<td>164.785</td>
</tr>
<tr>
<td></td>
<td>SF</td>
<td>4.48312</td>
<td>9.80864</td>
<td>22.8868</td>
</tr>
</tbody>
</table>

Table 3: Algorithm Speed (Characters Per Microsecond) in Large Alphabet Case on Three Systems
such as forward, single-step advances only. STL generic algorithms are specified to access sequences via iterators, which are generalizations of ordinary C/C++ pointers. STL defines five categories of iterators, the most powerful being random-access iterators, for which computing $i + n$ or $i - n$, for iterator $i$ and integer $n$, is a constant time operation. Forward iterators allow scanning a sequence with only single-step advances and only in a forward direction. AL and HAL require random access for most efficient operation of the skip loop, whereas Algorithm L, with only minor modifications to the expository version, can be made to work efficiently with forward iterators.

The efficiency issue is considered crucial in the STL approach to gener- icity. STL is not a set of specific software components but a set of requirements which components must satisfy. By making time complexity part of the requirements for components, STL ensures that compliant components not only have the specified interfaces and semantics but also meet certain computing time bounds. The requirements on most components are stated in terms of inputs and outputs that are linear sequences over some type $T$. The requirements are stated as generally as possible, but balanced against the goal of using efficient algorithms. In fact, the requirements were generally chosen based on knowledge of existing efficient concrete algorithms, by finding the weakest assumptions—about $T$ and about how the sequence elements are accessed—under which those algorithms could still be used without losing their efficiency. In most cases, the computing time requirements are stated as worst-case bounds, but exceptions are made when the concrete algorithms with the best worst-case bounds are not as good in the average case as other algorithms, provided the worst cases occur very infrequently in practice.

In the case of sequence search algorithms, the concrete algorithms considered for generalization to include in STL were various string-search algorithms, including BM, KMP, and SF. Although KMP has the lowest worst-case bound, it was stated in the original STL report [StepanovLee] that SF was superior in the average case. And although BM has excellent average time behavior, it was evidently ruled out as a generic algorithm because of its alphabet size dependency. Thus the generic search algorithm requirements were written with a $O(mn)$ time bound, to allow its implementation by SF.

Thus in the Draft C++ Standard dated December 1996 [DraftCPP], two sequence search functions are required, with the specifications:

```cpp
template <typename ForwardIterator1, typename ForwardIterator2>
```

3 The original STL requirements included the following statement (which has been dropped in more recent versions of the Draft C++ Standard): “...The Knuth-Morris-Pratt algorithm is not used here. While the KMP algorithm guarantees linear time, it tends to be slower in most practical cases than the naive algorithm with worst-case quadratic behavior ...”. As we have already seen from Table 1, however, a suitably optimized version of KMP—Algorithm L—is significantly faster than SF.

4 This did not preclude library implementors from also supplying a specialization of the search operation for the string search case, implemented with BM. The original requirements statement for the search operation noted this possibility but more recent drafts fail to mention it. We are not aware of any currently available STL implementations that do provide such a specialization.
**Effects:** Finds a subsequence of equal values in a sequence.

**Returns:** The first iterator \( i \) in the range \([\text{first1}, \text{last1} - (\text{last2} - \text{first2}))\) such that for any non-negative integer \( n \) less than \( \text{last2} - \text{first2} \) the following corresponding conditions hold: \( *(i + n) = *(\text{first2} + n) \), \( \text{pred}(*(i + n), *(\text{first2} + n)) \neq \text{false} \). Returns \text{last1} if no such iterator is found.

**Complexity:** At most \((\text{last1} - \text{first1}) \times (\text{last2} - \text{first2})\) applications of the corresponding predicate.

Before going further, we note that the results of the present article would allow the complexity requirement to be replaced with the much stronger requirement that the computing time be \( O((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) \).

We will base our discussion on the first interface, which assumes \text{operator==} is used for testing sameness of two sequence elements; the only added issue for the binary predicate case is the requirement mentioned earlier, that for HAL we must choose a hash function compatible with the binary predicate, in the sense that any two values that are equivalent according to the predicate must be mapped to the same value by the hash function. Fortunately, for a given predicate it is usually rather easy to choose a hash function that guarantees this property. (A heuristic guide is to choose a hash function that uses less information than the predicate.)

The fact that this standard interface only assumes forward iterators would seem to preclude HAL, since the skip loop requires random access. There are however many cases of sequence searching in which we do have random access, and we do not want to miss the speedup afforded by the skip loop in those cases. Fortunately, it is possible to provide for easy selection of the most appropriate algorithm under different actual circumstances, including whether random access or only forward access is available, and whether type \( T \) has a small or large number of distinct values. For this purpose we use traits, a programming device for compile-time selection of alternative type and code definitions. Traits are supported in C++ by the ability to give definitions of function templates or class templates for specializations of their template parameters.\(^5\)

\(^5\)Limited forms of the trait device were used in defining some iterator operations in the first implementations of STL. More recently the trait device has been adopted more
Algorithm L is not difficult to adapt to work with iterators instead of array indexing. The most straightforward translation would require random access iterators, but with a few adjustments we can express the algorithm entirely with forward iterator operations, making it fit the STL search function interface.

(User level search function 19a) ≡

```cpp
template <typename ForwardIterator1, typename ForwardIterator2>
inline ForwardIterator1 search(ForwardIterator1 text,
                                ForwardIterator1 textEnd,
                                ForwardIterator2 pattern,
                                ForwardIterator2 patternEnd)
{
    typedef iterator_traits<ForwardIterator1> T;
    return __search(text, textEnd, pattern, patternEnd, T::iterator_category());
}
```

Used in part 39a.

When we only have forward iterators, we use Algorithm L.

(Forward iterator case 19b) ≡

```cpp
template <typename ForwardIterator1, typename ForwardIterator2>
inline ForwardIterator1 __search_L(ForwardIterator1 text,
                                    ForwardIterator1 textEnd,
                                    ForwardIterator2 pattern,
                                    ForwardIterator2 patternEnd)
{
    typedef typename iterator_traits<ForwardIterator2>::difference_type Distance2;
    ForwardIterator1 advance, hold;
    ForwardIterator2 p, p1;
    Distance2 j, m;
    vector<Distance2> next;
    vector<ForwardIterator2> pattern_iterator;
    (Compute next table (C++ forward) 20a)
    m = next.size();
```
We store the next table in an STL vector, which provides random access to the integral next values; to be able to get from them back to the correct positions in the pattern sequence we also store iterators in another vector, `pattern_iterator`.

```cpp
template <typename ForwardIterator, typename Distance>
void compute_next(ForwardIterator pattern,
                  ForwardIterator patternEnd,
                  vector<Distance>& next,
                  vector<ForwardIterator>& pattern_iterator)
{
    Distance t = -1;
    next.reserve(32);
    pattern_iterator.reserve(32);
    next.push_back(-1);
    pattern_iterator.push_back(pattern);
    for (;;) {
        ForwardIterator advance = pattern;
        ++advance;
        if (advance == patternEnd)
            break;
        while (t >= 0 && *pattern != *pattern_iterator[t])
            t = next[t];
        ++pattern; ++t;
        if (*pattern == *pattern_iterator[t])
            next.push_back(next[t]);
        else
            next.push_back(t);
        pattern_iterator.push_back(pattern);
    }
}
```

Returning to the search algorithm itself, the details are as follows:
(Algorithm L, optimized linear pattern search (C++) 21a) ≡

(Handle pattern size = 1 as a special case (C++) 21b)
p1 = pattern; ++p1;
while (text != textEnd) {
  (Scan the text for a possible match (C++) 21c)
  (Verify whether a match is possible at the position found (C++) 21d)
  (Recover from a mismatch using the next table (C++ forward) 22a)
}
return textEnd;

Used in part 19b.

For the case of pattern size 1, we use the STL generic linear search algorithm, find.

(Handle pattern size = 1 as a special case (C++) 21b) ≡
if (next.size() == 1)
  return find(text, textEnd, *pattern);

Used in parts 21a, 42a, 46a.

The three parts of the body of the main loop are direct translations from the Ada versions given earlier, using pointer manipulation in place of array indexing.

(Scan the text for a possible match (C++) 21c) ≡
while (*text != *pattern)
  if (++text == textEnd)
    return textEnd;

Used in part 21a.

(Verify whether a match is possible at the position found (C++) 21d) ≡
p = p1; j = 1;
hold = text;
if (++text == textEnd)
  return textEnd;
while (*text == *p) {
  if (++p == patternEnd)
    return hold;
  if (++text == textEnd)
    return textEnd;
  ++j;
}

Used in part 21a.
(Recover from a mismatch using the next table (C++ forward) 22a) ≡

```c++
for (; ;) {
    j = next[j];
    if (j < 0) {
        ++text;
        break;
    }
    if (j == 0)
        break;
    p = pattern_iterator[j];
    while (*text == *p) {
        ++text; ++p; ++j;
        if (p == patternEnd) {
            (Compute and return position of match 22b)
        }
        if (text == textEnd)
            return textEnd;
    }
}
```

Used in part 21a.

Returning the match position requires use of the hold iterator saved for that purpose.

(Compute and return position of match 22b) ≡

```c++
advance = hold;
for (int i = m; --i >= 0;)
    ++advance;
while (advance != text)
    ++advance, ++hold;
return hold;
```

Used in part 22a.

Through the use of traits, we provide for automatic selection of either the above version of algorithm L in the case of forward or bidirectional iterators, or the faster HAL algorithm when random access to the sequences is available. STL random access iterators permit the use of either array index notation very similar that in the expository version of the algorithm, or pointer notation as shown above for algorithm L, but with additional operations such as $p + k$. Although it is commonplace to use pointer notation for efficiency reasons, we avoid it in this case because the calculation of the large value cannot be guaranteed to be valid in pointer arithmetic. The advantage of the single-test skip loop outweighs any disadvantage due to array notation calculations.

The trait interface also allows the user to supply the hash function, but various useful default hash functions can be provided. The full details, including complete source code, are shown in an appendix. The code is available from [http://www.cs.rpi.edu/~musser/gp](http://www.cs.rpi.edu/~musser/gp). The code supplied includes
a set of operation counting components \([\text{Mu96}]\) that permit easy gathering of statistics on many different kinds of operations, including data element accesses and comparisons, iterator operations, and “distance operations,” which are arithmetic operations on integer results of iterator subtractions. These counts are obtained without modifying the source code of the algorithms at all, by specializing their type parameters with classes whose operations have counters built into them. Table 4 shows counts of data comparisons and other data accesses, iterator “big jumps” and other iterator operations, and distance operations. In each case the counts are divided by the number of characters searched. These statistics come from searches of the same English text, \textit{Through the Looking Glass}, with the same selection of patterns, as discussed earlier. For ABM and TBM, not all operations were counted because the algorithms are from Hume and Sunday’s original C code and therefore could not be specialized with the counting components. For these algorithms a manually instrumented version (supplied as part of the code distribution \([\text{HS91}]\)) kept count of data comparisons and accesses. The table shows that HAL, like ABM and TBM, does remarkably few equality comparison operations on sequence elements—only about 1 per 100 elements for the longer patterns, no more than twice that for the shorter ones. They do access the elements substantially more often than that, in their respective skip loops, but still always sublinearly. With string matching, the comparisons and accesses are inexpensive, but in other applications of sequence matching they might cost substantially more than iterator or distance operations. In such applications the savings in execution time over SF or L could be even greater.

For example, an appendix shows one experiment in which the text of \textit{Through the Looking Glass} was stored as a sequence of words, each word being a character string, and the patterns were word sequences of different lengths chosen from evenly spaced positions in the target word sequence. In this case, element comparisons were word comparisons, which could be significantly more costly than iterator or distance operations. HAL was again substantially faster than the other contestants, SF and L. The ABM and TBM algorithms from \([\text{HS91}]\) were not considered because they are only applicable to string matching, but it was easy to specialize the three generic algorithms to this case of sequence matching, just by plugging in the appropriate types and, in the case of HAL, defining a suitable hash function. (We used a function that returns the first character of a word.)

8 How to Obtain the Appendices and Code

An expanded version of this paper, including appendices that contain and document the complete source code for all benchmark experiments described in the paper, will be maintained indefinitely for public access on the Internet at \url{http://www.cs.rpi.edu/~musser/gp/}. By downloading the Nuweb source file, \texttt{gensearch.w}, and using Briggs’ Nuweb tool \([\text{Briggs}]\),\footnote{We started with a version of the Nuweb tool previously modified by Ramsdell and Mengel and made additional small changes in terminology in the \TeX{} file the tool pro-} readers can also
easily generate all of the source code described in the paper and appendices.

9 Conclusion

When we began this research, our main goal was to develop a generic sequence search algorithm with a linear worst-case time bound and with better average case performance than KMP and SF, so that it could be used in generic software libraries such as the C++ Standard Template Library. We expected that for most of the useful special cases, such as English text or DNA substring matching, it would probably be better to provide separate algorithms tailored to those cases. It was therefore surprising to discover that for the substring matching problem itself a new, superior algorithm could be obtained by combining Boyer and Moore’s skip loop with the Knuth-Morris-Pratt algorithm. By also developing a hashed version of the skip loop and providing for selection of different variants of the technique using traits, we obtained a generic algorithm, HAL, with all of the attributes we originally sought. Moreover, when specialized to the usual string matching cases of the most practical interest, such as English text matching and DNA string matching, the new algorithm beats most of the existing string matching algorithms.

Since HAL has a linear upper bound on the number of comparisons, it can be used even in mission-critical applications where the potential $O(nm)$ behavior of the straightforward algorithm or Hume and Sunday’s TBM algorithm would be a serious concern. In such applications, as well as in less-critical applications, HAL’s performance in the average case is not only linear, but sublinear, beating even the best versions of the Boyer Moore algorithm. Since we have provided it in a generic form—in particular, in the framework of the C++ Standard Template Library—the new algorithm is easily reusable in many different contexts.

Acknowledgement: This work was partially supported by a grant from IBM Corporation.

References


[Briggs] P. Briggs, Nuweb, a simple literate programming tool, Version 0.87, 1989. 1, 8


duces: “part” is used in place of “scrap” and “definition” in place of “macro.” This version, called Nuweb 0.91, is available from http://www.cs.rpi.edu/~musser/gp/. The new version does not differ from previous versions in the way it produces code files from Nuweb source files.


A Tests of Expository Versions of the Algorithms

To help ensure against errors in the expository versions of the algorithms in this paper, we compiled them as part of several Ada test programs, using both the GNAT Ada 95 compiler, version 3.09, and the Aonix ObjectAda compiler, Special Edition, version 7.1.

A.1 Algorithm Declarations

We have not attempted to develop Ada generic subprograms based on the expository versions of the algorithms; instead we encapsulate them here with non-generic interfaces we can use in simple test programs based on string (Ada character array) searches.

(Sequence declarations 27a) ≡

| type Character_Sequence is array(Integer range <>) of Character; |
| type Integer_Sequence is array(Integer range <>) of Integer; |
| type Skip_Sequence is array(Character range <>) of Integer; |

Used in parts 31a, 34c, 36e.

(Algorithm subprogram declarations 27b) ≡

(Define procedure to compute next table 30a)
(Non-hashed algorithms 27c)
(Simple hash function declarations 28)
(HAL declaration 29a)

Used in parts 31a, 34c, 36e.

(Non-hashed algorithms 27c) ≡

function KMP(text, pattern: Character_Sequence;
  b, n, a, m: Integer) return Integer is
  pattern_size, j, k: Integer;
  next: Integer_Sequence(a .. m - 1);
begin
  (Compute next table 30b)
  (Algorithm L, optimized linear pattern search 3a)
end KMP;

function L(text, pattern: Character_Sequence;
  b, n, a, m: Integer) return Integer is
  pattern_size, j, k: Integer;
  next: Integer_Sequence(a .. m - 1);
begin
  pattern_size := m - a;
  (Compute next table 30b)
  (Algorithm L, optimized linear pattern search 3a)
end L;
function SF(text, pattern: Character_Sequence;
    b, n, a, m: Integer) return Integer is
    pattern_size, j, k: Integer;
begin
    pattern_size := m - a; k := b;
    (Handle pattern size = 1 as a special case 3b)
    while k <= n - pattern_size loop
        (Scan the text for a possible match 3c)
        (Verify whether a match is possible at the position found 4a)
        k := k - (j - a) + 1;
    end loop;
    return n;
end SF;

function AL(text, pattern: Character_Sequence;
    b, n, a, m: Integer) return Integer is
    pattern_size, text_size, j, k, large, adjustment, mismatch_shift: Integer;
    next: Integer_Sequence(a .. m - 1);
    skip: Skip_Sequence(Character'Range);
begin
    (Accelerated Linear algorithm 7b)
end AL;

Used in part 27b.

The following is a sample hash function definition that makes HAL essentially equivalent to AL.

(Simple hash function declarations 28) ≡
  subtype hash_range is Integer range 0..255;

  function hash(text: Character_Sequence; k: Integer) return hash_range;
  pragma inline(hash);

  function hash(text: Character_Sequence; k: Integer) return hash_range is
  begin
    return hash_range(character'pos(text(k)));
  end hash;

  suffix_size: constant Integer := 1;

Used in part 27b.
(HAL declaration 29a) ≡

function HAL(text, pattern: Character_Sequence;
    \text{b, n, a, m: Integer}) return Integer is
    \text{pattern_size, text_size, j, k, large, adjustment, mismatch_shift: Integer;}
    \text{next: Integer_Sequence(a .. m - 1);}
    \text{skip: Integer_Sequence(hash_range);}
begin
    \text{(Hashed Accelerated Linear algorithm 12b)}
end HAL;

Used in part 27b.

For comparison of HAL with other algorithms we also compose the following
declarations:

(Additional algorithms 29b) ≡

function AL0(text, pattern: Character_Sequence;
    \text{b, n, a, m: Integer}) return Integer is
    \text{pattern_size, j, k, d, mismatch_shift: Integer;}
    \text{next: Integer_Sequence(a .. m - 1);}
    \text{skip: Skip_Sequence(Character'Range);}
begin
    \text{(Accelerated Linear algorithm, preliminary version 5c)}
end AL0;

function SF1(text, pattern: Character_Sequence;
    \text{b, n, a, m: Integer}) return Integer is
    \text{pattern_size, j, k, k0: Integer;}
begin
    \text{pattern_size := m - a;}
    \text{if n < m then}
        \text{return n;}
    \text{end if;}
    j := a; k := b; k0 := k;
    \text{while j /= m loop}
        \text{if text(k) /= pattern(j) then}
            \text{if k = n - pattern_size then}
                \text{return n;}
            \text{else}
                \text{k0 := k0 + 1; k := k0; j := a;}
            \text{end if;}
        \text{else}
            \text{k := k + 1; j := j + 1;}
        \text{end if;}
    \text{end loop;}
    \text{return k0;}
end SF1;

function SF2(text, pattern: Character_Sequence;
    \text{b, n, a, m: Integer}) return Integer is
    \text{pattern_size, j, k, k0, n0: Integer;}
begin
    \text{pattern_size := m - a;}

if n - b < pattern_size then
  return n;
end if;
j := a; k := b; k0 := k; n0 := n - b;
while j /= m loop
  if text(k) = pattern(j) then
    k := k + 1; j := j + 1;
  else
    if n0 = pattern_size then
      return n;
    else
      k0 := k0 + 1; k := k0; j := a; n0 := n0 - 1;
    end if;
  end if;
end loop;
return k0;
end SF2;

Used in parts 31a, 34c, 36e.

For computing the KMP next table we provide the following procedure and calling code:

(Define procedure to compute next table 30a) ≡

procedure Compute_Next(pattern: Character_Sequence; a, m: Integer;
next: out Integer_Sequence) is
  j: Integer := a;
t: Integer := a - 1;
begins
  next(a) := a - 1;
  while j < m - 1 loop
    while t >= a and then pattern(j) /= pattern(t) loop
      t := next(t);
    end loop;
    j := j + 1; t := t + 1;
    if pattern(j) = pattern(t) then
      next(j) := next(t);
    else
      next(j) := t;
    end if;
  end loop;
end Compute_Next;

Used in part 27b.

(Compute next table 30b) ≡
Compute_Next(pattern, a, m, next);

Used in parts 5c, 7b, 12b, 27c.
A.2 Simple Tests

The first test program simply reads short test sequences from a file and reports the results of running the different search algorithms on them.

"Test_Search.adb" 31a ≡

```ada
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Io_Exceptions;
procedure Test_Search is
    ⟨Sequence declarations 27a⟩
    ⟨Variable declarations 31b⟩
    ⟨Algorithm subprogram declarations 27b⟩
    ⟨Additional algorithms 29b⟩
    ⟨Define procedure to read string into sequence 33a⟩
    ⟨Define procedure to output sequence 33b⟩
    ⟨Define algorithm enumeration type, names, and selector function 31d⟩
    ⟨Define Report procedure 33d⟩
    begin
        ⟨Set file small.txt as input file 31c⟩
        loop
            ⟨Read test sequences from file 32a⟩
            ⟨Run tests and report results 33c⟩
        end loop;
    end Test_Search;
```

(Variable declarations 31b) ≡

```ada
Comment, S1, S2: Character_Sequence(1 .. 100);
Base_Line, S1_Length, S2_Length, Last: Integer;
File: Text_Io.File_Type;
```

Used in part 31a.

(Set file small.txt as input file 31c) ≡

```ada
Text_Io.Open(File, Text_IO.In_File, "small.txt");
Text_Io.Set_Input(File);
```

Used in part 31a.

(Define algorithm enumeration type, names, and selector function 31d) ≡

```ada
type Algorithm_Enumeration is (Dummy, SF, SF1, SF2, L, AL, HAL);

Algorithm_Names: array(Algorithm_Enumeration) of String(1 .. 17) :=
("selection code ",
"SF ",
"HP SF ",
"SGI SF ",
```

31
function Algorithm(k: Algorithm_Enumeration;  
        text, pattern: Character_Sequence;  
        b, n, a, m: Integer) return Integer is  
begin  
case k is  
  when Dummy => return b;  
  when SF => return SF(text, pattern, b, n, a, m);  
  when SF1 => return SF1(text, pattern, b, n, a, m);  
  when SF2 => return SF2(text, pattern, b, n, a, m);  
  when L => return L(text, pattern, b, n, a, m);  
  when AL => return AL(text, pattern, b, n, a, m);  
  when HAL => return HAL(text, pattern, b, n, a, m);  
end case;  
end Algorithm;  

Test sequences are expected to be found in a file named small.txt. Each  
test set is contained on three lines, the first line being a comment or blank,  
the second line containing the text string to be searched, and the third the  
pattern to search for.

(Read test sequences from file 32a) ≡
  exit when Text_Io.End_Of_File;  
Get(Comment, Last);  
Put(Comment, Last); New_Line;  
  (Check for unexpected end of file 32b)  
Get(S1, Last);  
  (Check for unexpected end of file 32b)  
Put(“Text sequence: “); Put(S1, Last);  
S1_Length := Last;  
Get(S2, Last);  
Put(“Pattern sequence: “); Put(S2, Last);  
S2_Length := Last;  

Used in part 31a.

(Check for unexpected end of file 32b) ≡
  if Text_Io.End_Of_File then  
    Put_Line(“**** Unexpected end of file.”); New_Line;  
    raise Program_Error;  
  end if;  

Used in part 32a.
Define procedure to read string into sequence 33a

procedure Get(S: out Character_Sequence; Last: out Integer) is
    Ch: Character;
    I : Integer := 0;
begin
    while not Text_Io.End_Of_File loop
        Text_Io.Get_Immediate(Ch);
        I := I + 1;
        S(I) := Ch;
        exit when Text_Io.End_Of_Line;
    end loop;
    Last := I;
    Text_Io.Get_Immediate(Ch);
end Get;

Used in part 31a.

Define procedure to output sequence 33b

procedure Put(S: Character_Sequence; Last: Integer) is
begin
    for I in 1 .. Last loop
        Put(S(I));
    end loop;
    New_Line;
end Put;

Used in part 31a.

Run tests and report results 33c

Base_Line := 0;
for K in Algorithm_Enumeration'Succ(Algorithm_Enumeration'First) ..
    Algorithm_Enumeration'Last loop
    Put(" Using "); Put(Algorithm_Names(k)); New_Line;
    Report(K, S1, S2, 1, S1_Length + 1, 1, S2_Length + 1);
end loop;
New_Line;

Used in part 31a.

Define Report procedure 33d

procedure Report(K: Algorithm_Enumeration;
    S1, S2: Character_Sequence; b, n, a, m: Integer) is
    P: Integer;
begin
    P := Algorithm(K, S1, S2, b, n, a, m);
    Put(" String "); Put('"');
    (Output S2 34a)
    if P = n then
        Put(" not found");
    New_Line;
else
    Put('"'); Put(" found at position ");
    Put(P);
    New_Line;
end if;
if Base_Line = 0 then
    Base_Line := P - b;
else
    if P - b /= Base_Line then
        Put("*****Incorrect result!"); New_Line;
    end if;
    end if;
end Report;

Used in parts 31a, 34c.

(Output S2 34a) ≡

    for I in a .. m - 1 loop
        Put(S2(I));
    end loop;

Used in part 33d.

Here are a few small tests.

"small.txt" 34b ≡

    #
    Now's the time for all good men and women to come to the aid of their country.
    time
    #
    Now's the time for all good men and women to come to the aid of their country.
    timid
    #
    Now's the time for all good men and women to come to the aid of their country.
    try.
    # The following example is from the KMP paper.
    babcbabcaabcbabcabacbacabcbabcbababcab
    abcabcacab
    #
    aaaaaabcabcadefg
    abcad
    #
    aaaaaabcabcadefg
    ab

A.3 Large Tests

This Ada test program can read a long character sequence from a file and run extensive search tests on it. Patterns to search for, of a user-specified length, are selected from evenly-spaced positions in the long sequence.

"Test_Long_Search.adb" 34c ≡
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Test_Long_Search is
  F: Integer;
  Number_Of_Tests: Integer;
  Pattern_Size: Integer;
  Increment: Integer;
⟨Sequence declarations 27a⟩
⟨Data declarations 35a⟩
⟨Algorithm subprogram declarations 27b⟩
⟨Additional algorithms 29b⟩
⟨Define algorithm enumeration type, names, and selector function 31d⟩
⟨Define Report procedure 33d⟩
S2: Character_Sequence(0 .. 100);
begin
⟨Read test parameters 35c⟩
⟨Set file long.txt as input file 35b⟩
⟨Read character sequence from file 36a⟩
Increment := (S1_Length - S2_Length) / Number_Of_Tests;
⟨Run tests searching for selected subsequences 36b⟩
end Test_Long_Search;

⟨Data declarations 35a⟩ ≡
  Max_Size: constant Integer := 200_000;
  C: Character;
  S1: Character_Sequence(0 .. Max_Size);
  Base_Line, I, S1_Length, S2_Length: Integer;
  File: Text_Io.File_Type;

Used in parts 34c, 36e.

⟨Set file long.txt as input file 35b⟩ ≡
  Text_Io.Open(File, Text_IO.In_File, "long.txt");
  Text_Io.Set_Input(File);

Used in parts 34c, 36e.

⟨Read test parameters 35c⟩ ≡
  Put("Input Number of tests and pattern size: "); Text_Io.Flush;
  Get(Number_Of_Tests);
  Get(Pattern_Size);
  New_Line; Put("Number of tests: "); Put(Number_Of_Tests); New_Line;
  Put("Pattern size: "); Put(Pattern_Size); New_Line;
  S2_Length := Pattern_Size;

Used in parts 34c, 36e.
A.4 Timed Tests

This Ada test program reads a character sequence from a file and times searches for selected strings.

"Time_Long_Search.adb" 36e ≡
  with Text_Io; use Text_Io;
  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
  with Ada.Real_Time;
  procedure Time_Long_Search is

(Read character sequence from file 36a) ≡
  I := 0;
  while not Text_Io.End_Of_File loop
    Text_Io.Get_Immediate(C);
    S1(I) := C;
    I := I + 1;
  end loop;
  S1_Length := I;
  Put(S1_Length); Put(" characters read."); New_Line;

Used in parts 34c, 36e.

(Run tests searching for selected subsequences 36b) ≡
  F := 0;
  for K in 1 .. Number_Of_Tests loop
    (Select sequence S2 to search for in S1 36c)
    (Run tests 36d)
    end loop;

Used in part 34c.

(Select sequence S2 to search for in S1 36c) ≡
  for I in 0 .. Pattern_Size - 1 loop
    S2(I) := S1(F + I);
  end loop;
  F := F + Increment;

Used in parts 36b, 37b.

(Run tests 36d) ≡
  Base_Line := 0;
  for K in AlgorithmEnumeration'Succ(AlgorithmEnumeration'First) ..
    AlgorithmEnumeration'Last loop
    Put(" Using "); Put(AlgorithmNames(k)); New_Line;
    Report(K, S1, S2, 0, S1_Length, 0, S2_Length);
  end loop;
  New_Line;

Used in part 36b.
use Ada.Real_Time;
package My_Float is new Text_IO.Float_IO(Long_Float);
Base_Time: Long_Float;
Number_Of_Tests, Pattern_Size, Increment: Integer;
pragma Suppress(All_Checks);
(Sequence declarations 27a)
(Algorithm subprogram declarations 27b)
(Additional algorithms 29b)
(Define algorithm enumeration type, names, and selector function 31d)
(Data declarations 35a)
(Define run procedure 37b)
begin
(Read test parameters 35c)
(Set file long.txt as input file 35b)
(Read character sequence from file 36a)
Increment := (S1_Length - S2_Length) / Number_Of_Tests;
Base_Time := 0.0;
(Run and time tests searching for selected subsequences 37a)
end Time_Long_Search;

(Run and time tests searching for selected subsequences 37a) ≡
for K in Algorithm_Enumeration'Range loop
  Put("Timing "); Put(Algorithm_Names(K)); New_Line;
  Run(K, S1, S1_Length, S2_Length);
end loop;
New_Line;

Used in part 36e.

For a given algorithm, the Run procedure conducts a requested number of searches in sequence S1 for patterns of a requested size, selecting the patterns from evenly spaced positions in S1. It reports the total search length, time taken, and speed (total search length divided by time taken) of the searches.

(Define run procedure 37b) ≡
procedure Run(K: Algorithm_Enumeration;
  S1: Character_Sequence; Text_Size, Pattern_Size: Integer) is
  P, F: Integer;
  Start_Time, Finish_Time: Time;
  Total_Search: Integer;
  Time_Taken : Long_Float;
  S2: Character_Sequence(0 .. Pattern_Size - 1);
beginn
  F := 0;
  Total_Search := 0;
  Start_Time := Clock;
  for I in 1 .. Number_Of_Tests loop
    (Select sequence S2 to search for in S1 36c)
\[
P := \text{Algorithm}(K, S_1, S_2, 0, \text{Text}_\text{Size}, 0, \text{Pattern}_\text{Size});
\]
\[
\text{Total}_\text{Search} := \text{Total}_\text{Search} + P + \text{Pattern}_\text{Size};
\]
end loop;
\[
\text{Finish}_\text{Time} := \text{Clock};
\]
\{(Output \ statistics \ 38)\}
end Run;

Used in part 36e.

\{(Output \ statistics \ 38) \equiv \}
\[
\text{Time}_\text{Taken} := \text{Long}_\text{Float}(\text{(Finish}_\text{Time} - \text{Start}_\text{Time}) / \text{Milliseconds}(1)) / 1000.0 - \text{Base}_\text{Time};
\]
\[
\text{Put}("\text{Total}_\text{search} \text{length:} \ ");
\text{Put}(\text{Total}_\text{Search}); \text{Put}(" \text{bytes.}"); \text{New}_\text{Line};
\text{Put}("\text{Time:} \ "); \text{My}_\text{Float}.\text{Put}(\text{Time}_\text{Taken}, 5, 4, 0);
\text{Put}(" \text{seconds.}"); \text{New}_\text{Line};
\text{if} \ K \neq \text{Dummy} \text{then}
\text{Put}("\text{Speed:} \ ");
\text{My}_\text{Float}.\text{Put}(\text{Long}_\text{Float}(\text{Total}_\text{Search}) / 1.000.000.0 / \text{Time}_\text{Taken}, 5, 2, 0);
\text{Put}(" \text{MBytes/second.}"); \text{New}_\text{Line};
\text{else}
\text{Base}_\text{Time} := \text{Time}_\text{Taken};
\text{end if;}
\text{New}_\text{Line};

Used in part 37b.

\section{C++ Library Versions and Test Programs}

The code presented in this section is packaged in files that can be added to the standard C++ library and included in user programs with \texttt{#include} directives. (A few adjustments may be necessary depending on how well the target compiler conforms to the C++ standard.) With only minor changes, library maintainers should be able to incorporate the code into the standard library header files, replacing whatever search implementations they currently contain. The only significant work involved would be to construct the predicate versions of the search functions, which are not given here.
B.1 Generic Library Interfaces

B.1.1 Library Files

"new_search.h" 39a ≡
#ifndef NEW_SEARCH
# define NEW_SEARCH
#include <vector>
#include "search_traits.h"
#include <iterator>
using namespace std;

〈Define procedure to compute next table (C++) 44a〉
〈Define procedure to compute next table (C++ forward) 20b〉
〈User level search function 19a〉
〈Forward iterator case 19b〉
〈Bidirectional iterator case 40b〉
〈HAL with random access iterators, no trait passed 41a〉
〈User level search function with trait argument 41b〉
#endif

B.1.2 Search Traits

"search_traits.h" 39b ≡
#ifndef SEARCH_HASH_TRAITS
# define SEARCH_HASH_TRAITS
〈Generic search trait 39c〉
〈Search traits for character sequences 40a〉
#endif

The generic search trait class is used when there is no search trait specifically defined, either in the library or by the user, for the type of values in the sequences being searched, and when no search trait is explicitly passed to the search function.

〈Generic search trait 39c〉 ≡
    template <typename T>
    struct search_trait {
      enum {hash_range_max = 0};
      enum {suffix_size = 0};
      template <typename RandomAccessIterator>
      inline static unsigned int hash(RandomAccessIterator i) {
        return 0;
      }
    };

Used in part 39b.

The “hash” function used in this trait maps everything to 0; it would be a source of poor performance if it were actually used in the HAL algo-
algorithm. In fact it is not, because the code in the search function checks for
suffix_size = 0 and uses algorithm L in that case. This definition of hash
permits compilation to succeed even if the compiler fails to recognize that
the code segment containing the call of hash is dead code.

For traditional string searches, the following specialized search traits are
provided:

(\textit{Search traits for character sequences 40a}) \equiv
\begin{verbatim}
template <> struct search_trait<signed char> {
    enum {hash_range_max = 256};
    enum {suffix_size = 1};
    template <typename RandomAccessIterator>
    inline static unsigned int hash(RandomAccessIterator i) {
        return *i;
    }
};

typedef unsigned char unsigned_char;
template <> struct search_trait<unsigned char> {
    enum {hash_range_max = 256};
    enum {suffix_size = 1};
    template <typename RandomAccessIterator>
    inline static unsigned int hash(RandomAccessIterator i) {
        return *i;
    }
};
\end{verbatim}

Used in part 39b.

\subsection{Search Functions}

The main user-level search function interface and an auxiliary function
\_\_search\_L for the forward iterator case were given in the body of the
paper. With bidirectional iterators we again use the forward iterator ver-

(\textit{Bidirectional iterator case 40b}) \equiv
\begin{verbatim}
template <typename BidirectionalIterator1, typename BidirectionalIterator2>
inline BidirectionalIterator1 __search(BidirectionalIterator1 text,
BidirectionalIterator1 textEnd,
BidirectionalIterator2 pattern,
BidirectionalIterator2 patternEnd,
bidirectional_iterator_tag)
{
    return __search\_L(text, textEnd, pattern, patternEnd);
}
\end{verbatim}

Used in part 39a.

When we have random access iterators and no search trait is passed as an ar-
argument, we use a search trait associated with V = RandomAccessIterator1::value_type
to obtain the hash function and related parameters. Then we use the user-
level search function that takes a search trait argument and uses HAL. If
no search trait has been specifically defined for type $V$, then the generic
search_hash_trait is used, causing the search_hashed algorithm to re-
sort to algorithm L.

\begin{verbatim}
(HAL with random access iterators, no trait passed 41a) ≡
template <typename RandomAccessIterator1, typename RandomAccessIterator2>
inline RandomAccessIterator1 __search(RandomAccessIterator1 text,
    RandomAccessIterator1 textEnd,
    RandomAccessIterator2 pattern,
    RandomAccessIterator2 patternEnd,
    random_access_iterator_tag)
{
    typedef iterator_traits<RandomAccessIterator1>::value_type V;
    typedef search_trait<V> Trait;
    return search_hashed(text, textEnd, pattern, patternEnd,
        static_cast<Trait*>(0));
}
\end{verbatim}

Used in part 39a.

Finally, we have a user-level search function for the case of random access
iterators and an explicitly passed search trait.

\begin{verbatim}
(User level search function with trait argument 41b) ≡
template <typename RandomAccessIterator1, typename RandomAccessIterator2,
    typename Trait>
RandomAccessIterator1 search_hashed(RandomAccessIterator1 text,
    RandomAccessIterator1 textEnd,
    RandomAccessIterator2 pattern,
    RandomAccessIterator2 patternEnd,
    Trait*)
{
    typedef typename iterator_traits<RandomAccessIterator1>::difference_type
        Distance1;
    typedef typename iterator_traits<RandomAccessIterator2>::difference_type
        Distance2;
    if (pattern == patternEnd) return text;
    Distance2 pattern_size, j, m;
    pattern_size = patternEnd - pattern;
    if (Trait::suffix_size == 0 || pattern_size < Trait::suffix_size)
        return __search_L(text, textEnd, pattern, patternEnd);
    Distance1 i, k, large, adjustment, mismatch_shift, text_size;
    vector<Distance1> next, skip;
    (Hashed Accelerated Linear algorithm (C++) 42a)
}
\end{verbatim}

Used in part 39a.
The C++ version of HAL is built from parts corresponding to those expressed in Ada in the body of the paper. Note that in place of `text(n + k)` we can write `textEnd + k` for the location and `textEnd[k]` for the value at that location.

(Hashed Accelerated Linear algorithm (C++) 42a) ≡

\[
\begin{align*}
& k = 0; \\
& text\_size = textEnd - text; \quad \langle \text{Compute next table (C++) 44b} \rangle \\
& \langle \text{Handle pattern size = 1 as a special case (C++) 21b} \rangle \\
& \langle \text{Compute skip table and mismatch shift using the hash function (C++) 43c} \rangle \\
& \text{large} = text\_size + 1; \\
& \text{adjustment} = \text{large} + \text{pattern\_size} - 1; \\
& \text{skip}[\text{Trait::hash(pattern + pattern\_size - 1)}] = \text{large}; \\
& k -= text\_size; \\
& \text{for}(;;) { \\
& \hspace{1em} k += \text{pattern\_size} - 1; \\
& \hspace{1em} \text{if} \ (k >= 0) \text{ break}; \\
& \hspace{1em} \langle \text{Scan the text using a single-test skip loop with hashing (C++) 42b} \rangle \\
& \hspace{1em} \langle \text{Verify match or recover from mismatch (C++) 42c} \rangle \\
& } \\
& \text{return textEnd;}
\end{align*}
\]

Used in part 41b.

(Scan the text using a single-test skip loop with hashing (C++) 42b) ≡

\[
\begin{align*}
& \text{do} \{ \\
& \hspace{1em} k += \text{skip}[\text{Trait::hash(textEnd + k)}]; \\
& \} \text{ while} \ (k < 0); \\
& \text{if} \ (k < \text{pattern\_size}) \\
& \hspace{1em} \text{return textEnd; } \\
& \text{k -= adjustment;}
\end{align*}
\]

Used in part 42a.

(Verify match or recover from mismatch (C++) 42c) ≡

\[
\begin{align*}
& \text{if} \ (\text{textEnd}[k] \neq \text{pattern}[0]) \\
& \hspace{1em} k += \text{mismatch\_shift}; \\
& \text{else} \{ \\
& \hspace{1em} \langle \text{Verify the match for positions 1 through pattern\_size - 1 (C++) 43a} \rangle \\
& \hspace{1em} \text{if} \ (\text{mismatch\_shift} > j) \\
& \hspace{2em} k += \text{mismatch\_shift} - j; \\
& \hspace{1em} \text{else} \\
& \hspace{2em} \langle \text{Recover from a mismatch using the next table (C++) 43b} \rangle \\
& \} \\
\end{align*}
\]

Used in parts 42a, 46a.
Verify the match for positions 1 through pattern_size - 1 (C++)

\[
\begin{align*}
\text{j} & = 1; \\
\text{for (;;) \{} \\
\text{++k;} & \\
\text{if (textEnd[k] \neq pattern[j])} & \text{break;} \\
\text{++j;} & \\
\text{if (j == pattern_size)} & \text{return textEnd + k - pattern_size + 1;} \\
\text{\}} \\
\end{align*}
\]

Used in part 42c.

Recover from a mismatch using the next table (C++)

\[
\begin{align*}
\text{for (;;) \{} \\
\text{j} & = \text{next}[j]; \\
\text{if (j < 0)} & \text{break;} \\
\text{\}} \\
\text{if (j == 0)} & \text{break;} \\
\text{while (textEnd[k] == pattern[j]) \{} \\
\text{++k;} & \text{++j;} \\
\text{if (j == pattern_size)} & \text{return textEnd + k - pattern_size;} \\
\text{\}} \\
\text{if (k == 0)} & \text{return textEnd;} \\
\text{\}} \\
\end{align*}
\]

Used in part 42c.

B.1.4 Skip Table Computation

\[
\begin{align*}
\text{m} & = \text{next.size();} \\
\text{for (i = 0; i < Trait::hash_range_max; ++i)} & \text{skip.push_back(m - Trait::suffix_size + 1);} \\
\text{for (j = Trait::suffix_size - 1; j < m - 1; ++j)} & \text{skip[Trait::hash(pattern + j)] = m - 1 - j;} \\
\text{mismatch_shift} & = \text{skip[Trait::hash(pattern + m - 1)];} \\
\text{skip[Trait::hash(pattern + m - 1)]} & = 0; \\
\end{align*}
\]

Used in part 42a.

B.1.5 Next Table Procedure and Call

When we have random access to the pattern, we take advantage of it in computing the \text{next} table (we do not need to create the \text{pattern_iterator}
table used in the forward iterator version).

\begin{verbatim}
(Define procedure to compute next table (C++) 44a) ≡

\begin{verbatim}
template <typename RandomAccessIterator, typename Distance>
void compute_next(RandomAccessIterator pattern,
                  RandomAccessIterator patternEnd,
                  vector<Distance>& next)
{
    Distance pattern_size = patternEnd - pattern, j = 0, t = -1;
    next.reserve(32);
    next.push_back(-1);
    while (j < pattern_size - 1) {
        while (t >= 0 && pattern[j] != pattern[t])
            t = next[t];
        ++j; ++t;
        if (pattern[j] == pattern[t])
            next.push_back(next[t]);
        else
            next.push_back(t);
    }
}
\end{verbatim}

Used in part 39a.

(Compute next table (C++) 44b) ≡

\begin{verbatim}
compute_next(pattern, patternEnd, next);
\end{verbatim}

Used in parts 42a, 46a.

\section*{B.2 Experimental Version for Large Alphabet Case}

For comparison with HAL in the large alphabet case we also implemented
the experimental version that uses a large skip table and no hashing, as
described in the body of the paper.

"experimental_search.h" 44c ≡

\begin{verbatim}
(Experimental search function with skip loop without hashing 44d) ≡

#include <vector>
using namespace std;

struct large_alphabet_trait {
    typedef unsigned short T;
    enum {suffix_size = 1};
    enum {hash_range_max = (1u << (sizeof(T) * 8)) - 1};
};
\end{verbatim}

In our experiments, we assume that the element type is a 2-byte unsigned short.

(Experimental search function with skip loop without hashing 44d) ≡

\begin{verbatim}
#include <vector>
using namespace std;

struct large_alphabet_trait {
    typedef unsigned short T;
    enum {suffix_size = 1};
    enum {hash_range_max = (1u << (sizeof(T) * 8)) - 1};
};
\end{verbatim}

44
template <> struct search_trait<unsigned short> {
    enum {hash_range_max = 256};
    enum {suffix_size = 1};
    template <typename RandomAccessIterator>
    inline static unsigned int hash(RandomAccessIterator i) {
        return (unsigned char)(*i);
    }
};

template <typename T>
class skewed_value {
    static T skew;
    T value;
public:
    skewed_value() : value(0) {} 
    skewed_value(T val) : value(val - skew) {} 
    operator T () { return value + skew; }
    static void setSkew(T askew) { skew = askew; }
    void clear() { value = 0; }
};

template <typename T> T skewed_value<T>::skew;

template <typename T, typename RandomAccessIterator, int size>
class skewed_array {
    typedef skewed_value<T> value_type;
    static value_type array[size];
    RandomAccessIterator pattern, patternEnd;
public:
    skewed_array(T skew, RandomAccessIterator pat, RandomAccessIterator patEnd):
        pattern(pat),patternEnd(patEnd){ value_type::setSkew(skew); }
    ~skewed_array() {
        while (pattern != patternEnd)
            array[*pattern++].clear();
    }
    value_type operator[](int index) const { return array[index]; }
    value_type& operator[](int index) { return array[index]; }
};

template <typename T, typename RandomAccessIterator, int size>
skewed_value<T> skewed_array<T,RandomAccessIterator,size>::array[size];

template <typename RandomAccessIterator1, typename RandomAccessIterator2>
RandomAccessIterator1 search_no_hashing(RandomAccessIterator1 text,
    RandomAccessIterator1 textEnd,
    RandomAccessIterator2 pattern,
    RandomAccessIterator2 patternEnd)
{
    typedef typename iterator_traits<RandomAccessIterator1>::difference_type Distance1;
    typedef typename iterator_traits<RandomAccessIterator2>::difference_type Distance2;
    typedef large_alphabet_trait Trait;
    if (pattern == patternEnd)
        return text;
    Distance1 k, text_size, large, adjustment, mismatch_shift;
}
Distance2 j, m, pattern_size;
pattern_size = patternEnd - pattern;
if (pattern_size < Trait::suffix_size)
    return __search_L(text, textEnd, pattern, patternEnd);
vector<Distance1> next;
skewed_array<Distance1, RandomAccessIterator2, Trait::hash_range_max+1>
    skip(pattern_size - Trait::suffix_size + 1, pattern, patternEnd);

(Accelerated Linear algorithm, no hashing (C++) 46a)
}

Used in part 44c.

(Accelerated Linear algorithm, no hashing (C++) 46a) ≡
    k = 0;
text_size = textEnd - text;
(Compute next table (C++) 44b)
(Handle pattern size = 1 as a special case (C++) 21b)
(Compute skip table and mismatch shift, no hashing (C++) 46b)
large = text_size + 1;
adjustment = large + pattern_size - 1;
skip[*(pattern + m - 1)] = large;

k -= text_size;
for (;;) {
    k += pattern_size - 1;
    if (k >= 0) break;
(Scan the text using a single-test skip loop, no hashing (C++) 47a)
(Verify match or recover from mismatch (C++) 42c)
}
return textEnd;

Used in part 44d.

(Compute skip table and mismatch shift, no hashing (C++) 46b) ≡
m = next.size();
for (j = Trait::suffix_size - 1; j < m - 1; ++j)
    skip[*(pattern + j)] = m - 1 - j;
mismatch_shift = skip[*(pattern + m - 1)];
    skip[*(pattern + m - 1)] = 0;

Used in part 46a.
(Scan the text using a single-test skip loop, no hashing (C++) 47a) ≡

```c++
do {
    k += skip[*(textEnd + k)];
} while (k < 0);
if (k < pattern_size)
    return textEnd;
k -= adjustment;
```

Used in part 46a.

### B.3 DNA Search Functions and Traits

The following definitions are for use in DNA search experiments. Four different search functions are defined using 2, 3, 4, or 5 characters as arguments to hash functions.

"DNA_search.h" 47b ≡

(Define DNA search traits 48)

```c++
template <typename RandomAccessIterator1, typename RandomAccessIterator2>
inline RandomAccessIterator1 hal2(RandomAccessIterator1 text,
RandomAccessIterator1 textEnd,
RandomAccessIterator2 pattern,
RandomAccessIterator2 patternEnd)
{
    return search_hashed(text, textEnd, pattern, patternEnd,
static_cast<search_trait_dna2*>(0));
}

template <typename RandomAccessIterator1, typename RandomAccessIterator2>
inline RandomAccessIterator1 hal3(RandomAccessIterator1 text,
RandomAccessIterator1 textEnd,
RandomAccessIterator2 pattern,
RandomAccessIterator2 patternEnd)
{
    return search_hashed(text, textEnd, pattern, patternEnd,
static_cast<search_trait_dna3*>(0));
}

template <typename RandomAccessIterator1, typename RandomAccessIterator2>
inline RandomAccessIterator1 hal4(RandomAccessIterator1 text,
RandomAccessIterator1 textEnd,
RandomAccessIterator2 pattern,
RandomAccessIterator2 patternEnd)
{
    return search_hashed(text, textEnd, pattern, patternEnd,
static_cast<search_trait_dna4*>(0));
}

template <typename RandomAccessIterator1, typename RandomAccessIterator2>
inline RandomAccessIterator1 hal5(RandomAccessIterator1 text,
RandomAccessIterator1 textEnd,
RandomAccessIterator2 pattern,
RandomAccessIterator2 pattern,
RandomAccessIterator2 patternEnd)
{
    return search_hashed(text, textEnd, pattern, patternEnd,
static_cast<search_trait_dna5*>(0));
}
```
RandomAccessIterator2 patternEnd)
{
    return search_hashed(text, textEnd, pattern, patternEnd,
                        static_cast<search_trait_dna5*>(0));
}

(Define DNA search traits 48) ≡
struct search_trait_dna2 {
    enum {hash_range_max = 64};
    enum {suffix_size = 2};
    template <typename RAI>
    inline static unsigned int hash(RAI i) {
        return (*(i-1) + (*(i) << 3)) & 63;
    }
};

struct search_trait_dna3 {
    enum {hash_range_max = 512};
    enum {suffix_size = 3};
    template <typename RAI>
    inline static unsigned int hash(RAI i) {
        return (*i-2) + (*i-1) << 3) + (*i) << 6)) & 511;
    }
};

struct search_trait_dna4 {
    enum {hash_range_max = 256};
    enum {suffix_size = 4};
    template <typename RAI>
    inline static unsigned int hash(RAI i) {
        return (*i-3) + (*i-2) << 2) + (*i-1) << 4) + (*i) << 6)) & 255;
    }
};

struct search_trait_dna5 {
    enum {hash_range_max = 256};
    enum {suffix_size = 5};
    template <typename RAI>
    inline static unsigned int hash(RAI i) {
        return (*i-4) + (*i-3) << 2) + (*i-2) << 4) + (*i-1) << 6) + (*i) << 8)) & 255;
    }
};

Used in part 47b.

B.4 Simple Tests

In the test programs we want to compare the new search functions with the existing search function from an STL algorithm library implementation, so we rename the existing one.
As in the Ada version of the code, the first test program simply reads short test sequences from a file and reports the results of running the different search algorithms on them.

"test_search.cpp" 49b ≡

```cpp
#include <iostream>
#include <fstream>
#include "new_search.h"
#include "hume.hh"
#include "DNA_search.h"
using namespace std;
int Base_Line;
```

```cpp
define algorithms header with existing search function renamed 49a
#include <algorithm>
define __search __stl_search
#include <algorithm>
define search stl_search
#define __search __stl_search
#include <algorithm>
define search stl_search
#define __search __stl_search
```

Used in parts 49b, 52, 55b, 58, 59d, 63b, 65b.

enum algorithmEnumeration {
    Dummy, SF, L, HAL, ABM, TBM, GBM, HAL2, HAL3, HAL4, HAL5
};

const char* algorithmNames[] = {
    "selection code", "SF", "L", "HAL", "ABM", "TBM", "GBM",
    "HAL2", "HAL3", "HAL4", "HAL5"
};

#ifndef DNA_TEST
algorithmEnumeration alg[] = {Dummy, SF, L, HAL, ABM, TBM};
const char textFileName[] = "long.txt";
```

49
const char wordFileName[] = "words.txt";
#else
    algorithm Enumeration alg[] = {Dummy, SF, L, HAL, ABM, GBM,
                                 HAL2, HAL3, HAL4, HAL5};
#endif
const char textFileName[] = "dnatext.txt";
const char wordFileName[] = "dnaword.txt";
#endif

const int number_of_algorithms = sizeof(alg)/sizeof(alg[0]);

template <typename Container, typename Container__const_iterator>
inline void
    Algorithm(int k, const Container& x, const Container& y,
              Container__const_iterator& result)
{
    switch (alg[k]) {
    case Dummy:
        // does nothing, used for timing overhead of test loop
        result = x.begin(); return;
    case SF:
        result = stl_search(x.begin(), x.end(), y.begin(), y.end()); return;
    case L:
        result = __search_L(x.begin(), x.end(), y.begin(), y.end()); return;
    case HAL:
        result = search(x.begin(), x.end(), y.begin(), y.end()); return;
    case ABM:
        result = fbm(x.begin(), x.end(), y.begin(), y.end()); return;
    case TBM:
        result = hume(x.begin(), x.end(), y.begin(), y.end()); return;
    case GBM:
        result = gdbm(x.begin(), x.end(), y.begin(), y.end()); return;
    case HAL2:
        result = hal2(x.begin(), x.end(), y.begin(), y.end()); return;
    case HAL3:
        result = hal3(x.begin(), x.end(), y.begin(), y.end()); return;
    case HAL4:
        result = hal4(x.begin(), x.end(), y.begin(), y.end()); return;
    case HAL5:
        result = hal5(x.begin(), x.end(), y.begin(), y.end()); return;
    }
    result = x.begin(); return;
}

Used in parts 49b, 52, 55b.

(Read test sequences from file (C++) 50) ≡
get(ifs, Comment);
if (ifs.eof())
    break;
    copy(Comment.begin(), Comment.end(), out); cout << endl;
get(ifs, S1);
( Check for unexpected end of file (C++) 51a)
cout << "Text string:......";
copy(S1.begin(), S1.end(), out);
cout << endl;

get(ifs, S2);
\{Check for unexpected end of file (C++) 51a\}
cout << "Pattern string:....";
copy(S2.begin(), S2.end(), out); cout << endl;

Used in part 49b.

\{Check for unexpected end of file (C++) 51a\} ≡
if (ifs.eof()) {
  cout << "**** Unexpected end of file." << endl;
  exit(1);
}

Used in part 50.

\{Define procedure to read string into sequence (C++) 51b\} ≡
template <typename Container>
void get(istream& is, Container& S) {
  S.erase(S.begin(), S.end());
  char ch;
  while (is.get(ch)) {
    if (ch == '\n')
      break;
    S.push_back(ch);
  }
}

Used in part 49b.

\{Run tests and report results (C++) 51c\} ≡
Base_Line = 0;
for (int k = 1; k < number_of_algorithms; ++k) {
  cout << "Using " << algorithm_names[k] << ":" << endl;
  Report(algorithm_enumeration(k), S1, S2, separator);
}
cout << endl;

Used in parts 49b, 54d, 55a.

\{Define Report procedure (C++) 51d\} ≡
template <typename Container>
void Report(algorithm_enumeration k, const Container& S1,
const Container& S2, const char* separator)
{
  typename Container::const_iterator P;
  typedef typename Container::value_type value_t;

  51
Algorithm(k, S1, S2, P);
cout << " String " << '"
; copy(S2.begin(), S2.end(), ostream_iterator<value_t>(cout, separator));
if (P == S1.end())
cout << '"
, << " not found" << endl;
else
cout << '"
, << " found at position " << P - S1.begin() << endl;
if (Base_Line == 0)
Base_Line = P - S1.begin();
else
if (P - S1.begin() != Base_Line)
cout << "*****Incorrect result!" << endl;
}

B.5 Large Tests

The following program for conducting tests on a long text sequence performs the same tests as the Ada version, plus searches for words of the requested pattern size selected from a given dictionary (which is read from a file).

"test_long_search.cpp" 52 ≡

#include "new_search.h"
#include "hume.hh"
#include "DNA_search.h"
#include <iterator>
#include <vector>
#include <map>
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

typedef unsigned char data;
typedef vector<data> sequence;
sequence S1, S2;

int Base_Line;
unsigned int Number_Of_Tests, Number_Of_Pattern_Sizes, Increment;

int main()
{
    unsigned int F, K, j;
    (Read test parameters (C++) 53a)
    (Read dictionary from file, placing words of size j in dictionary[j] 53b)
    (Read character sequence from file (C++) 54b)
    for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
        (Trim dictionary[Pattern_Size[j]] to have at most Number_Of_Tests words 54a)
Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
cerr << Pattern_Size[j] << " " << flush;
const char* separator = "";
\textit{Output header (C++) 54c}
\textit{Run tests searching for selected subsequences (C++) 54d}
\textit{Run tests searching for dictionary words (C++) 55a}
}
}

\textit{(Read test parameters (C++) 53a) ≡}

\begin{verbatim}
cout << "Input number of tests (for each pattern size): " << flush;
cin >> Number_Of_Tests;
cout << "Input number of pattern sizes: " << flush;
cin >> Number_Of_Pattern_Sizes;
cout << "Input pattern sizes: " << flush;
vector<int> Pattern_Size(Number_Of_Pattern_Sizes);
for (j = 0; j < Number_Of_Pattern_Sizes; ++j)
    cin >> Pattern_Size[j];
cout << "\nNumber of tests: " << Number_Of_Tests << endl;
cout << "Pattern sizes: ";
for (j = 0; j < Number_Of_Pattern_Sizes; ++j)
    cout << Pattern_Size[j] << " ";
cout << endl;
\end{verbatim}

Used in parts 52, 55b, 58, 59d, 63b, 65b.

\textit{(Read dictionary from file, placing words of size j in dictionary[j] 53b) ≡}

\begin{verbatim}
ifstream dictfile(wordFileName);
typedef istream_iterator<string> string_input;
typedef map<int, vector<sequence>, less<int>> map_type;
map_type dictionary;
sequence S;
string S0;
string_input si(dictfile);
while (si != string_input()) {
    S0 = *si++;
    sequence S(S0.begin(), S0.end());
    dictionary[S.size()].push_back(S);
}
\end{verbatim}

Used in parts 52, 55b, 59d.
Trim dictionary[Pattern_Size[j]] to have at most Number_Of Tests words 54a) ≡

```cpp
vector<sequence>& diction = dictionary[Pattern_Size[j]]; 
if (diction.size() > Number_Of_Tests) {
    vector<sequence> temp;
    int Skip_Amount = diction.size() / Number_Of_Tests;
    for (unsigned int T = 0; T < Number_Of_Tests; ++T) {
        temp.push_back(diction[T * Skip_Amount]);
    }
    diction = temp;
}
```

Used in parts 52, 55b, 59d.

Read character sequence from file (C++) 54b) ≡

```cpp
ifstream ifs(textFileName);
char C;
while (ifs.get(C))
    S1.push_back(C);
cout << S1.size() << " characters read." << endl;
```

Used in parts 52, 55b, 59d.

Output header (C++) 54c) ≡

```cpp
cout << "\n\n-----------------------------------------------------------\n"
   << "Searching for patterns of size " << Pattern_Size[j]
   << " ...");
cout << "(" << Number_Of_Tests << " patterns from the text, 
   " << dictionary[Pattern_Size[j]].size() << " from the dictionary)" << endl;
```

Used in parts 52, 55b, 59d, 63b, 65b.

Run tests searching for selected subsequences (C++) 54d) ≡

```cpp
F = 0;
for (K = 1; K <= Number_Of_Tests; ++K) {
    (Select sequence S2 to search for in S1 (C++) 54e)
    (Run tests and report results (C++) 51c)
}
```

Used in parts 52, 63b.

Select sequence S2 to search for in S1 (C++) 54e) ≡

```cpp
S2.erase(S2.begin(), S2.end());
copy(S1.begin() + F, S1.begin() + F + Pattern_Size[j], back_inserter(S2));
F += Increment;
```

Used in part 54d.
Run tests searching for dictionary words (C++) 55a ≡

\[
\text{for } (K = 0; K < \text{dictionary[Pattern\_Size[j]].size(); ++K) } \{ \\
\quad S2 = \text{dictionary[Pattern\_Size[j]][K];} \\
\quad \text{(Run tests and report results (C++) 51c)} \\
\}
\]

Used in part 52.

B.6 Timed Tests

Again, the following program for timing searches conducts the same searches as in the Ada version, plus searches for words of the requested pattern size selected from a given dictionary.

"time_long_search.cpp" 55b ≡

(Include algorithms header with existing search function renamed 49a)

```cpp
#include "new_search.h"
#include "hume.hh"
#include "DNA_search.h"
#include <iterator>
#include <deque>
#include <vector>
#include <map>
#include <iostream>
#include <fstream>
#include <ctime>
#include <string>
using namespace std;
```

typedef unsigned char data;
typedef vector<data> sequence;
sequence S1;
int Base_Line;
unsigned int Number_Of_Tests, Number_Of_Pattern_Sizes, Increment;
double Base_Time = 0.0;
(Define algorithm enumeration type, names, and selector function (C++) 49c)
(Define run procedure (C++ forward) 56a)

```cpp
int main()
{
    int j;
    (Read test parameters (C++) 53a)
    (Read character sequence from file (C++) 54b)
    (Read dictionary from file, placing words of size j in dictionary[j] 53b)
    for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
        (Trim dictionary[Pattern\_Size[j]] to have at most Number\_Of\_Tests words 54a)
        Increment = (S1.size() - Pattern\_Size[j]) / Number_Of_Tests;
        (Output header (C++) 54c)
        cerr << Pattern\_Size[j] << " " << flush;
        (Run and time tests searching for selected subsequences (C++) 57a)
    }
}
```

55
The following test procedure is programmed using forward iterator operations only, so that it can be applied to a non-random access container (e.g., list), assuming the designated algorithm works with forward iterators.

```
(Define run procedure (C++ forward) 56a) ≡
template <typename Container>
void Run(int k, const Container& S1, const vector<Container>& dictionary, int Pattern_Size)
{
    typename Container::const_iterator P;
    int F = 0, d, K;
    double Start_Time, Finish_Time, Time_Taken;
    long Total_Search = 0;
    Start_Time = clock();
    Container S2;
    for (K = 1; K <= Number_Of_Tests; ++K) {
        typename Container::const_iterator u = S1.begin();
        advance(u, F);
        S2.erase(S2.begin(), S2.end());
        for (int I = 0; I < Pattern_Size; ++I)
            S2.push_back(*u++);
        F += Increment;
        (Run algorithm and record search distance 56b)
    }
    for (K = 0; K < dictionary.size(); ++K) {
        S2 = dictionary[K];
        (Run algorithm and record search distance 56b)
    }
    Finish_Time = clock();
    (Output statistics (C++) 57b)
}
```

Used in parts 55b, 58, 65b.

```
(Run algorithm and record search distance 56b) ≡
Algorithm(k, S1, S2, P);
    d = 0;
    distance(S1.begin(), P, d);
    Total_Search += d + Pattern_Size;
```

Used in parts 56a, 62.
Run and time tests searching for selected subsequences (C++) 57a ≡

\[
\text{Base\_Time} = 0.0; \\
\text{for (int } k = 0; k < \text{number\_of\_algorithms}; ++k) { \\
\hspace{1em} \text{if } (k != 0) \\
\hspace{2em} \text{cout } \ll \text{"Timing } \ll \text{algorithm\_names[k] } \ll \text{" } \ll \text{endl; } \\
\hspace{2em} \text{Run}(k, S1, \text{dictionary[Pattern\_Size[j]]}, \text{Pattern\_Size[j]}); \\
\hspace{1em} } \text{cout } \ll \text{endl;} \\
\}
\]

Used in parts 55b, 58, 59d, 65b.

Output statistics (C++) 57b ≡

\[
\text{Time\_Taken} = (\text{Finish\_Time} - \text{Start\_Time})/\text{CLOCKS\_PER\_SEC} - \text{Base\_Time}; \\
\text{if } (k == 0) \text{ Base\_Time = Time\_Taken;} \\
\text{else } \{ \\
\hspace{1em} \text{cout } \ll \text{"Total search length: } \ll \text{Total\_Search } \ll \text{elements } \ll \text{endl; } \\
\hspace{1em} \text{cout } \ll \text{"Time: } \ll \text{Time\_Taken } \ll \text{seconds. } \ll \text{endl; } \\
\hspace{1em} \text{double Speed = Time\_Taken == 0.0 ? 0.0 : static\_cast<double>(Total\_Search) / 1000000 / Time\_Taken; } \\
\hspace{1em} \text{cout } \ll \text{"Speed: } \ll \text{Speed } \ll \text{elements/microsecond. } \ll \text{endl } \ll \text{endl; } \\
\}
\]

Used in part 56a.

B.7 Timed Tests (Large Alphabet)

Again, the following program for timing searches conducts the same searches as in the Ada version, plus searches for words of the requested pattern size selected from a given dictionary.

Define algorithm enumeration type, names, and selector function (C++ large alphabet) 57c ≡

\[
\text{enum algorithm\_enumeration} \{ \\
\hspace{1em} \text{Dummy, SF, L, HAL, NHAL } \\
\}; \\
\text{const char* algorithm\_names[]} = \{ \\
\hspace{1em} \text{"selection code", "SF", "L", "HAL", "NHAL" } \\
\}; \\
\]

\[
\text{const int number\_of\_algorithms} = 5; \\
\]

\[
\text{template } <\text{typename Container, typename Container\_const\_iterator}> \\
\text{inline void Algorithm(int k, const Container& x, const Container& y, Container\_const\_iterator& result) } \\
\{
\text{switch (algorithm\_enumeration(k)) }\{ \\
\hspace{1em} \text{case Dummy: } \\
\hspace{2em} \text{// does nothing, used for timing overhead of test loop } \\
\hspace{2em} \text{result } = x.\text{begin(); return;} \\
\}
\]

57
case SF:
    result = stl_search(x.begin(), x.end(), y.begin(), y.end()); return;
case L:
    result = __search_L(x.begin(), x.end(), y.begin(), y.end()); return;
case HAL:
    result = search(x.begin(), x.end(), y.begin(), y.end()); return;
case NHAL:
    result = search_no_hashing(x.begin(), x.end(), y.begin(), y.end()); return;
}
result = x.begin(); return;
}

Used in part 58.

"experimental_search.cpp" 58 ≡

#include "new_search.h"
#include "experimental_search.h"
#include <iterator>
#include <deque>
#include <vector>
#include <map>
#include <iostream>
#include <fstream>
#include <ctime>
using namespace std;

typedef unsigned short data;
typedef vector<data> sequence;

sequence S1;

int Base_Line, Number_Of_Tests, Number_Of_Pattern_Sizes, Increment;
double Base_Time = 0.0;

int main()
{
    int j;
    for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
        Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
        cerr << Pattern_Size[j] << " " << flush;
    }
    cerr << endl;
Define RandomNumberGenerator class 59a

```cpp
int random(int max_value) { return rand() % max_value; }

template <int MAX_VALUE> struct RandomNumberGenerator {
    int operator() () { return random(MAX_VALUE); }
};
```

Used in part 58.

Generate data sequence 59b

```cpp
generate_n(back_inserter(S1), 100000, RandomNumberGenerator<65535>());
```

Used in part 58.

Generate dictionary 59c

```cpp
typedef map<int, vector<sequence>, less<int>> map_type;
map_type dictionary;
for(int i = 0; i < Number_Of_Pattern_Sizes; ++i) {
    int pattern_size = Pattern_Size[i];
    for(int j = 0; j < Number_Of_Tests; ++j) {
        int position = random(S1.size() - pattern_size);
        dictionary[pattern_size].push_back(sequence);
        copy(S1.begin() + position, S1.begin() + position + pattern_size,
             back_inserter(dictionary[pattern_size].back()));
    }
}
```

Used in part 58.

B.8 Counted Tests

The following program runs the same searches as in the timing program, but in addition to times it records and reports counts of many different types of operations, including equality comparisons on data, iterator operations, and “distance operations,” which are arithmetic operations on integer results of iterator subtractions. These counts are obtained without modifying the source code of the algorithms at all, by specializing their type parameters with classes whose operations have counters built into them.

"count_long_search.cpp" 59d

```cpp
#include "new_search.h"
#include "hume.hh"
#include <iterator>
#include <vector>
#include <map>
```
```cpp
#include <iostream>
#include <fstream>
#include <ctime>
#include <string>
using namespace std;

typedef vector<data> sequence;
sequence S1;
int Base_Line;
unsigned int Number_Of_Tests, Number_Of_Pattern_Sizes, Increment;
double Base_Time = 0.0;

recorder<> stats[number_of_algorithms];

int main()
{
    int j;
    for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
        Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
        cerr << Pattern_Size[j] << " " << flush;
        (Run and time tests searching for selected subsequences (C++) 57a)
    }
    cerr << endl;
}

}}
```

```
#include <utility>
using namespace std;
namespace std { namespace rel_ops {} };
using namespace std::rel_ops;
#include "counters.h"

typedef unsigned char basedata;

typedef value_counter<basedata> data;
typedef iterator_counter<vector<data>::iterator> citer;

struct search_trait_for_counting {
    enum {hash_range_max = 256};
    enum {suffix_size = 1};
    inline static unsigned int hash(const citer& i) {return (*i).base();}
};
```
(Define algorithm enumeration type, names, and selector function (C++ counter) 61) \equiv

```c++
enum algorithm_enumeration {
    Dummy, STL_search, L, HAL, ABM, TBM
};
const char* algorithm_names[] = {
    "selection code", "SF", "L", "HAL", "ABM", "TBM"
};
#ifndef LIST_TEST
const int number_of_algorithms = 6;
#else
const int number_of_algorithms = 3;
#endif
const char textFileName[] = "long.txt";
const char wordFileName[] = "words.txt";

template <typename Container, typename Container__const_iterator>
void Algorithm(int k, const Container& x, const Container& y,
    Container__const_iterator& result)
{
    switch (algorithm_enumeration(k)) {
    case Dummy:
        result = x.begin(); // does nothing, used for timing overhead of test loop
        return;
    case STL_search:
        result = stl_search(citer(x.begin()), citer(x.end()),
            citer(y.begin()), citer(y.end())).base();
        return;
    case L:
        result = __search_L(citer(x.begin()), citer(x.end()),
            citer(y.begin()), citer(y.end())).base();
        return;
    #ifndef LIST_TEST
    case HAL:
        result = search_hashed(citer(x.begin()), citer(x.end()),
            citer(y.begin()), citer(y.end()),
            static_cast<search_trait_for_counting*>(0)).base();
        return;
    case ABM:
        fbmprep((const basedata*)y.begin(), y.size());
        result = (typename Container::const_iterator)
            fbmexec_cnt((const basedata*)x.begin(), x.size());
        data::accesses += ::pat.accs;
        data::equal_comparisons += ::pat.cmps;
        return;
    case TBM:
        humprep((const basedata*)y.begin(), y.size());
        result = (typename Container::const_iterator)
```
humexec_cnt((const basedata*)x.begin(), x.size());
data::accesses += ::pat.accs;
data::equal_comparisons += ::pat.cmps;
result = result;
return;
#endif
}
result = x.begin();
return;
}

Used in part 59d.

Define run procedure (C++ counter) 62 ≡

template <typename Container>
void Run(int k, const Container & S1,
         const vector<Container> & dictionary, int Pattern_Size)
{
    typename Container::const_iterator P;
    int F = 0, K, d;
    double Start_Time, Finish_Time, Time_Taken;
    long Total_Search = 0;
    stats[k].reset();
    Start_Time = clock();
    Container S2;
    for (K = 1; K <= Number_Of_Tests; ++K) {
        typename Container::const_iterator u = S1.begin();
        advance(u, F);
        S2.erase(S2.begin(), S2.end());
        for (int I = 0; I < Pattern_Size; ++I)
            S2.push_back(*u++);
        F += Increment;
        (Run algorithm and record search distance 56b)
    }
    for (K = 0; K < dictionary.size(); ++K) {
        S2 = dictionary[K];
        (Run algorithm and record search distance 56b)
    }
    stats[k].record();
    Finish_Time = clock();
    (Output statistics (C++ counter) 63a)
}

Used in part 59d.
Output statistics (C++ counter) 63a

\[
\text{Time}_\text{Taken} = (\text{Finish}_\text{Time} - \text{Start}_\text{Time})/\text{CLOCKS}_\text{PER}_\text{SEC} - \text{Base}_\text{Time};
\]

if \( k == 0 \)
\[
\text{Base}_\text{Time} = \text{Time}_\text{Taken};
\]
else {
// \text{data::report}(\text{cout, Total}_\text{Search, 4});
// \text{citer::report}(\text{cout, Total}_\text{Search, 4});
// \text{cdistance::report}(\text{cout, Total}_\text{Search, 4});
\text{stats}[k].\text{hide}_\text{all}();
\text{stats}[k].\text{show}(\text{recorder<>::ASSIGNMENT, true});
\text{stats}[k].\text{show}(\text{recorder<>::LESS}_\text{ THAN, true});

\text{cout} \ll \text{"Total search length: \"} \ll \text{Total}_\text{Search} \ll \text{\" elements\"} \ll \text{endl;}
\text{cout} \ll \text{"Time: \"} \ll \text{Time}_\text{Taken} \ll \text{\" seconds.\"} \ll \text{endl;}
\text{double Speed = Time}_\text{Taken} == 0.0 ? 0.0 : 
\text{(double)Total}_\text{Search} / 1000000 / \text{Time}_\text{Taken;}
\text{cout} \ll \text{"Speed: \"} \ll \text{Speed} \ll \text{\" elements/microsecond.\"} \ll \text{endl} \ll \text{endl;}
}

Used in part 62.

B.9 Application to Matching Sequences of Words

B.9.1 Large Tests

This C++ program specializes the generic search functions to work with sequences of words (character strings). It reads a text file in as a sequence of words, and for each of a specified set of pattern sizes, it searches for word sequences of that size selected from evenly spaced positions in the target sequence. These searches are the counterpart of the first kind of searches done in the previous programs on character sequences; the dictionary word searches of the previous programs are omitted here.

"test_word_search.cpp" 63b

\include "new_search.h"
\include <iterator>
\include <vector>
\include <map>
\include <iostream>
\include <fstream>
\include <string>
using namespace std;

typedef string data;
typedef vector<data> sequence;

sequence S1, S2;
int Base_Line, Number_Of_Tests, Number_Of_Pattern_Sizes, Increment;
\include "search trait for word searches 64a"
\include "algorithm enumeration type, names, and selector function (C++ word) 64b"
\include "Report procedure (C++) 51d"
int main()
{
    int F, K, j;

    // Read test parameters (C++) 53a
    typedef map<int, vector<sequence>, less<int>> map_type;
    map_type dictionary;

    // Read word sequence from file (C++) 65a
    cout << S1.size() << " words read." << endl;
    const char* separator = " ";

    // Run tests searching for selected subsequences (C++) 54d
    for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
        Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
        // Output header (C++) 54c
    }
}

For a hash function the program uses a mapping of a word to its first character. Although this would not be a good hash function in hashed associative table lookup, it works satisfactorily here because there is less need for uniformity of hash value distribution.

// Define search trait for word searches 64a ≡
struct search_word_trait {
    typedef vector<string>::const_iterator RAI;
    enum {hash_range_max = 256};
    enum {suffix_size = 1};
    inline static unsigned int hash(RAI i) {
        return (*i)[0];
    }
};

// Define algorithm enumeration type, names, and selector function (C++ word) 64b ≡
enum algorithm_enumeration {
    Dummy, STL_search, L, HAL
};
const char* algorithm_names[] = {
    "selection code", "SF", "L", "HAL"
};
#ifndef LIST_TEST
const int number_of_algorithms = 4;
#else
const int number_of_algorithms = 3;
#endif

template<typename Container, typename Container__const_iterator>
inline void
    Algorithm(int k, const Container& x, const Container& y,
Container__const_iterator& result) {
    switch (algorithm_enumeration(k)) {
        case Dummy:
            result = x.begin(); return; // does nothing, used for timing overhead of test loop
        case STL_search:
            result = stl_search(x.begin(), x.end(), y.begin(), y.end()); return;
        case L:
            result = __search_L(x.begin(), x.end(), y.begin(), y.end()); return;
        #ifndef LIST_TEST
        case HAL:
            result = search_hashed(x.begin(), x.end(), y.begin(), y.end(),
                                    (search_word_trait*)0); return;
        #endif
    }
    result = x.begin(); return;
}

(Read word sequence from file (C++) 65a) ≡

    ifstream ifs("long.txt");
    typedef istream_iterator<string> string_input;
    copy(string_input(ifs), string_input(), back_inserter(S1));

Used in parts 63b, 65b.

B.9.2 Timed Tests

We also omit the dictionary searches in the following program which times
searches for selected subsequences, in this case by defining a map from ints
to empty dictionaries (in order to reuse some of the previous code).

"time_word_search.cpp" 65b ≡

    #include "new_search.h"
    #include <iterator>
    #include <vector>
    #include <map>
    #include <iostream>
    #include <fstream>
    #include <string>
    #include <ctime>
    // #include <list>
    // #define LIST_TEST
    using namespace std;

typedef string data;
typedef vector<data> sequence;

    sequence S1, S2;
    int Base_Line, Number_Of_Tests, Number_Of_Pattern_Sizes, Increment;
    double Base_Time = 0.0;
int main()
{
    int j;
    typedef map<int, vector<sequence >, less<int> > map_type;
    map_type dictionary;
    cout << S1.size() << " words read." << endl;
    for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
        Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
    }
}

C Index of Part Names

(Accelerated Linear algorithm, no hashing (C++) 46a) Referenced in part 44d.
(Accelerated Linear algorithm, preliminary version 5c) Referenced in part 29b.
(Accelerated Linear algorithm 7b) Referenced in part 27c.
(Additional algorithms 29b) Referenced in parts 31a, 34c, 36e.
(Algorithm L, optimized linear pattern search (C++) 21a) Referenced in part 19b.
(Algorithm L, optimized linear pattern search 3a) Referenced in part 27c.
(Algorithm subprogram declarations 27b) Referenced in parts 31a, 34c, 36e.
(Basic KMP 2) Referenced in part 27c.
(Bidirectional iterator case 40b) Referenced in part 39a.
(Check for unexpected end of file (C++) 51a) Referenced in part 50.
(Compute and return position of match 22b) Referenced in part 22a.
(Compute next table (C++ forward) 20a) Referenced in part 19b.
(Compute next table (C++) 44b) Referenced in parts 42a, 46a.
(Compute next table 30b) Referenced in parts 5c, 7b, 12b, 27c.
(Compute skip table and mismatch shift using the hash function (C++) 43c) Referenced in part 42a.
(Compute skip table and mismatch shift using the hash function 12a) Referenced in part 12b.
(Compute skip table and mismatch shift, no hashing (C++) 46b) Referenced in part 46a.
(Compute skip table and mismatch shift 6a) Referenced in parts 5c, 7b.
(Data declarations 35a) Referenced in parts 34c, 36e.
(Define DNA search traits 48) Referenced in part 47b.
(Define RandomNumberGenerator class 59a) Referenced in part 58.
(Define Report procedure (C++) 51d) Referenced in parts 49b, 52, 63b.
(Define Report procedure 33d) Referenced in parts 31a, 34c.
(Define algorithm enumeration type, names, and selector function (C++ counter) 61) Referenced in part 59d.
(Define algorithm enumeration type, names, and selector function (C++ large al-
Recover from a mismatch using the next table, with k translated

(Define algorithm enumeration type, names, and selector function (C++ word) 64b)
Referred in parts 63b, 65b.

(Define algorithm enumeration type, names, and selector function (C++) 49c) Referred in parts 49b, 52, 55b.

(Define algorithm enumeration type, names, and selector function 31d) Referred in parts 31a, 34c, 36c.

(Define procedure to compute next table (C++ forward) 20b) Referred in part 39a.
(Define procedure to compute next table (C++) 44a) Referred in part 39a.
(Define procedure to compute next table 30a) Referred in part 27b.
(Define procedure to output sequence 33b) Referred in part 31a.
(Define procedure to read string into sequence (C++) 51b) Referred in part 49b.
(Define procedure to read string into sequence 33a) Referred in part 31a.
(Define run procedure (C++ counter) 62) Referred in part 59d.
(Define run procedure (C++ forward) 56a) Referred in parts 55b, 58, 65b.
(Define run procedure 37b) Referred in part 36e.
(Define search trait for word searches 64a) Referred in parts 63b, 65b.
(Define types needed for counting operations 60) Referred in part 59d. (Experimental search function with skip loop without hashing 44d) Referred in part 44c.

(Forward iterator case 19b) Referred in part 39a.
(Generate data sequence 59b) Referred in part 58.
(Generate dictionary 59c) Referred in part 58.
(Generic search trait 39c) Referred in part 39b.
(HAL declaration 29a) Referred in part 27b.
(HAL with random access iterators, no trait passed 41a) Referred in part 39a.
(Handle pattern size = 1 as a special case (C++) 21b) Referred in parts 21a, 42a, 46a.

(Handle pattern size = 1 as a special case 3b) Referred in parts 3a, 5c, 7b, 12b, 27c.
(Hashed Accelerated Linear algorithm (C++) 42a) Referred in part 41b.
(Hashed Accelerated Linear algorithm 12b) Referred in part 29a.
(Include algorithms header with existing search function renamed 49a) Referred in parts 49b, 52, 55b, 58, 59d, 63b, 65b.
(Non-hashed algorithms 27c) Referred in part 27b.
(Output S2 34a) Referred in part 33d.
(Output header (C++) 54c) Referred in parts 52, 55b, 58, 59d, 63b, 65b.
(Output statistics (C++ counter) 63a) Referred in part 62.
(Output statistics (C++) 57b) Referred in part 56a.
(Output statistics 38) Referred in part 37b.
(Read character sequence from file (C++) 54b) Referred in parts 52, 55b, 59d.
(Read character sequence from file 36a) Referred in parts 34c, 36c.
(Read dictionary from file, placing words of size j in dictionary[j] 53b) Referred in parts 52, 55b, 59d.
(Read test parameters (C++) 53a) Referred in parts 52, 55b, 58, 59d, 63b, 65b.
(Read test parameters 35c) Referred in parts 34c, 36c.
(Read test sequences from file (C++) 50) Referred in part 49b.
(Read test sequences from file 32a) Referred in part 31a.
(Read word sequence from file (C++) 65a) Referred in parts 63b, 65b.
(Recover from a mismatch using the next table (C++ forward) 22a) Referred in part 21a.
(Recover from a mismatch using the next table (C++) 43b) Referred in part 42c.
(Recover from a mismatch using the next table, with k translated 8c) Referred in part 8a.
(Recover from a mismatch using the next table 4b) Referred in parts 3a, 5c.
(Run algorithm and record search distance 56b) Referenced in parts 56a, 62.
(Run and time tests searching for selected subsequences (C++) 57a) Referenced in parts 55b, 58, 59d, 65b.
(Run and time tests searching for selected subsequences 37a) Referenced in part 36e.
(Run tests and report results (C++) 51c) Referenced in parts 49b, 54d, 55a.
(Run tests and report results 33c) Referenced in part 31a.
(Run tests searching for dictionary words (C++) 55a) Referenced in part 52.
(Run tests searching for selected subsequences (C++) 54d) Referenced in parts 52, 63b.
(Run tests searching for selected subsequences 36b) Referenced in part 34c.
(Run tests 36d) Referenced in part 36b.
(Scan the text for a possible match (C++) 21c) Referenced in part 21a.
(Scan the text for a possible match 3c) Referenced in parts 3a, 27c.
(Scan the text using a single-test skip loop with hashing (C++) 42b) Referenced in part 42a.
(Scan the text using a single-test skip loop with hashing 11) Referenced in part 12b.
(Scan the text using a single-test skip loop, no hashing (C++) 47a) Referenced in part 46a.
(Scan the text using a single-test skip loop, with k translated 7a) Referenced in part 7b.
(Scan the text using a single-test skip loop 6b) Not referenced.
(Scan the text using the skip loop 5a) Referenced in part 5c.
(Select sequence S2 to search for in S1 (C++) 54e) Referenced in part 54d.
(Select sequence S2 to search for in S1 36c) Referenced in parts 36b, 37b.
(Sequence declarations 27a) Referenced in parts 31a, 34c, 36e.
(Set file long.txt as input file 35b) Referenced in parts 34c, 36e.
(Set file small.txt as input file 31c) Referenced in part 31a.
(Simple hash function declarations 28) Referenced in part 27b.
(Trim dictionary[Pattern_Size[j]] to have at most Number_Of_Tests words 54a) Referenced in parts 52, 55b, 59d.
(Verify match or recover from mismatch (C++) 42c) Referenced in parts 42a, 46a.
(Verify match or recover from mismatch 8a) Referenced in parts 7b, 12b.
(Verify the match for positions 1 through pattern_size - 1 (C++) 43a) Referenced in part 42c.
(Verify the match for positions a + 1 through m - 1, with k translated 8b) Referenced in part 8a.
(Verify whether a match is possible at the position found (C++) 21d) Referenced in part 21a.
(Verify whether a match is possible at the position found 4a) Referenced in parts 3a, 27c.
<table>
<thead>
<tr>
<th>Pattern Size</th>
<th>Algorithm</th>
<th>Comparisons</th>
<th>Other Accesses</th>
<th>Big Jumps</th>
<th>Other Iter Ops</th>
<th>Distance Ops</th>
<th>Total Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SF</td>
<td>1.036</td>
<td>0.001</td>
<td>0.000</td>
<td>4.192</td>
<td>2.002</td>
<td>7.231</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>1.028</td>
<td>0.001</td>
<td>0.000</td>
<td>4.095</td>
<td>0.177</td>
<td>5.301</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>0.018</td>
<td>0.513</td>
<td>0.551</td>
<td>1.104</td>
<td>2.431</td>
<td>4.617</td>
</tr>
<tr>
<td></td>
<td>ABM</td>
<td>0.017</td>
<td>0.528</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>0.021</td>
<td>0.511</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>SF</td>
<td>1.034</td>
<td>0.000</td>
<td>0.000</td>
<td>4.170</td>
<td>2.002</td>
<td>7.203</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>1.031</td>
<td>0.000</td>
<td>0.000</td>
<td>4.098</td>
<td>0.159</td>
<td>5.288</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>0.013</td>
<td>0.266</td>
<td>0.291</td>
<td>0.583</td>
<td>0.658</td>
<td>1.811</td>
</tr>
<tr>
<td></td>
<td>ABM</td>
<td>0.013</td>
<td>0.277</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>0.014</td>
<td>0.266</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>6</td>
<td>SF</td>
<td>1.042</td>
<td>0.000</td>
<td>0.000</td>
<td>4.211</td>
<td>2.000</td>
<td>7.254</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>1.037</td>
<td>0.000</td>
<td>0.000</td>
<td>4.119</td>
<td>0.194</td>
<td>5.350</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>0.011</td>
<td>0.189</td>
<td>0.211</td>
<td>0.422</td>
<td>0.482</td>
<td>1.315</td>
</tr>
<tr>
<td></td>
<td>ABM</td>
<td>0.012</td>
<td>0.198</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>0.012</td>
<td>0.189</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>SF</td>
<td>1.048</td>
<td>0.000</td>
<td>0.000</td>
<td>4.243</td>
<td>2.000</td>
<td>7.291</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>1.042</td>
<td>0.000</td>
<td>0.000</td>
<td>4.135</td>
<td>0.220</td>
<td>5.396</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>0.010</td>
<td>0.150</td>
<td>0.170</td>
<td>0.339</td>
<td>0.392</td>
<td>1.060</td>
</tr>
<tr>
<td></td>
<td>ABM</td>
<td>0.011</td>
<td>0.157</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>0.011</td>
<td>0.150</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>10</td>
<td>SF</td>
<td>1.052</td>
<td>0.000</td>
<td>0.000</td>
<td>4.263</td>
<td>2.000</td>
<td>7.315</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>1.044</td>
<td>0.000</td>
<td>0.000</td>
<td>4.142</td>
<td>0.233</td>
<td>5.418</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>0.009</td>
<td>0.126</td>
<td>0.144</td>
<td>0.289</td>
<td>0.337</td>
<td>0.905</td>
</tr>
<tr>
<td></td>
<td>ABM</td>
<td>0.010</td>
<td>0.132</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>0.010</td>
<td>0.126</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>14</td>
<td>SF</td>
<td>1.077</td>
<td>0.000</td>
<td>0.000</td>
<td>4.384</td>
<td>2.000</td>
<td>7.460</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>1.060</td>
<td>0.000</td>
<td>0.000</td>
<td>4.197</td>
<td>0.328</td>
<td>5.585</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>0.010</td>
<td>0.105</td>
<td>0.125</td>
<td>0.250</td>
<td>0.305</td>
<td>0.796</td>
</tr>
<tr>
<td></td>
<td>ABM</td>
<td>0.010</td>
<td>0.109</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>0.011</td>
<td>0.105</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>18</td>
<td>SF</td>
<td>1.105</td>
<td>0.000</td>
<td>0.000</td>
<td>4.525</td>
<td>2.000</td>
<td>7.629</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>1.077</td>
<td>0.000</td>
<td>0.000</td>
<td>4.257</td>
<td>0.436</td>
<td>5.770</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>0.011</td>
<td>0.096</td>
<td>0.117</td>
<td>0.234</td>
<td>0.295</td>
<td>0.753</td>
</tr>
<tr>
<td></td>
<td>ABM</td>
<td>0.010</td>
<td>0.099</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>TBM</td>
<td>0.011</td>
<td>0.096</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 4: Average Number of Operations Per Character in English Text Searches