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

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 Σ. 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 `next`, defined by

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

(We let $\mathtt{next}(j) = a - 1$ 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:[1]

[1] 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 5⟩ ≡

```
pattern_size := m - a; j := a; k := b;
while j < m and then k < n loop
  while j >= a and then text(k) /= pattern(j) loop
    j := next(j);
  end loop;
  k := k + 1; j := j + 1;
end loop;
if j = m then
  return k - pattern_size;
else
  return n;
end if;
```

Used in part 53b.

A return value i between b and $n -$ `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(k)` is usually not equal to `pattern(j)` and `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 6⟩ ≡
     pattern_size := m - a; k := b;
     ⟨Handle pattern size = 1 as a special case 7a⟩
     while k <= n - pattern_size loop
        ⟨Scan the text for a possible match 7b⟩
        ⟨Verify whether a match is possible at the position found 8⟩
        ⟨Recover from a mismatch using the next table 9⟩
     end loop;
     return n;
```

Used in part 53b.

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

⟨Handle pattern size = 1 as a special case 7a⟩ ≡

```
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 6, 12, 16, 25, 53b.

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

⟨Scan the text for a possible match 7b⟩ ≡

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

Used in parts 6, 53b.

⟨Verify whether a match is possible at the position found 8⟩ ≡

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

Used in parts 6, 53b.

⟨Recover from a mismatch using the next table 9⟩ ≡

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

Used in parts 6, 12.

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:          ......uuuuuuuuuua....  ......uuuuuuuuuue....
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 `skip` function, which maps any character of the alphabet to the range $[0, m-a]$ and is defined as follows:

$$\texttt{skip}(x) = \begin{cases} m-a & \text{if } \forall j : a \leq j < m \Rightarrow p_j \neq x \\ m-1-i & \text{otherwise, where } i = \max\{j : a \leq j < m \wedge p_j = x\} \end{cases}$$

This is the same function as Boyer and Moore's δ_1 [BM77]. The following code replaces the scan part of Algorithm L:

⟨Scan the text using the skip loop 11a⟩ ≡

```
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 12.

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 11b⟩ ≡

```
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 12.

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 12⟩ ≡
    pattern_size := m - a; k := b;
    ⟨Handle pattern size = 1 as a special case 7a⟩
    ⟨Compute next table 59⟩
    ⟨Compute skip table and mismatch shift 13⟩
    while k <= n - pattern_size loop
      ⟨Scan the text using the skip loop 11a⟩
      ⟨Verify the match for positions a through m - 2 11b⟩
      if mismatch_shift > j - a then
        k := k + (mismatch_shift - (j - a));
      else
        ⟨Recover from a mismatch using the next table 9⟩
      end if;
    end loop;
    return n;
```

Used in part 56b.

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

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

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

⟨Compute skip table and mismatch shift 13⟩ ≡

```
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;
```

Used in parts 12, 16.

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.

⟨Scan the text using a single-test skip loop 14⟩ ≡

```
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;
```

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`.

⟨Scan the text using a single-test skip loop, with k translated 15⟩ ≡

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

Used in part 16.

This saves an instruction over testing `k >= 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

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

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

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

Hence it suffices to choose $\mathtt{large} = n - b + 1$.

```
⟨Accelerated Linear algorithm 16⟩ ≡
    pattern_size := m - a; text_size := n - b; k := b;
    ⟨Handle pattern size = 1 as a special case 7a⟩
    ⟨Compute next table 59⟩
    ⟨Compute skip table and mismatch shift 13⟩
    large := text_size + 1;
    skip(pattern(m - 1)) := large;
    adjustment := large + pattern_size - 1;
    k := k - n;
    loop
      k := k + pattern_size - 1;
      exit when k >= 0;
      ⟨Scan the text using a single-test skip loop, with k translated 15⟩
      ⟨Verify match or recover from mismatch 17⟩
    end loop;
    return n;
```

Used in part 53b.

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 17⟩ ≡
    if text(n + k) /= pattern(a) then
      k := k + mismatch_shift;
    else
      ⟨Verify the match for positions a + 1 through m - 1, with k translated 18⟩
      if mismatch_shift > j - a then
        k := k + (mismatch_shift - (j - a));
      else
        ⟨Recover from a mismatch using the next table, with k translated 19⟩
      end if;
    end if;
```

Used in parts 16, 25.

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 18⟩ ≡

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

Used in part 17.

⟨Recover from a mismatch using the next table, with k translated 19⟩ ≡

```
loop
  j := next(j);
  if j < a then
     k := k + 1;
     exit;
  end if;
  exit when j = a;
  while text(n + k) = pattern(j) loop
    k := k + 1; j := j + 1;
    if j = m then
      return n + k - pattern_size;
    end if;
    if k = 0 then
      return n;
    end if;
  end loop;
end loop;
```

Used in part 17.

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 O^2 with MIPS R5000 2.1 processor;
3. Apogee apCC compiler, version 3.0, 200 MHz UltraSPARC processor.

Pattern Size	Algorithm	System 1	System 2	System 3
2	ABM	8.89665	24.6946	**32.9261**
	HAL	8.26117	24.6946	**32.9261**
	L	6.08718	24.6946	**32.9261**
	SF	4.28357	9.87784	24.6946
	TBM	**10.5142**	**32.9261**	**32.9261**
4	ABM	20.4425	46.7838	68.9446
	HAL	**23.3995**	**51.0369**	**83.6137**
	L	6.52724	27.8712	38.9093
	SF	4.29622	9.84923	23.3919
	TBM	21.2602	49.123	71.4517
6	ABM	28.1637	60.2832	89.4829
	HAL	**31.2569**	**63.6323**	**108.055**
	L	6.45279	27.4015	37.9265
	SF	4.28142	9.84005	22.1973
	TBM	29.2294	62.249	93.8837
8	ABM	33.7463	69.2828	106.674
	HAL	**37.0999**	**73.0482**	**126.801**
	L	6.34086	26.6684	36.5241
	SF	4.23323	9.78229	22.0342
	TBM	35.3437	72.2627	112.007
10	ABM	39.6329	76.2308	117.47
	HAL	**42.5986**	**80.5134**	**135.202**
	L	6.32525	26.6383	36.1904
	SF	4.22537	9.74924	21.9134
	TBM	41.1973	78.7439	125.714
14	ABM	47.7986	89.1214	129.631
	HAL	**49.8997**	**92.9962**	**147.511**
	L	6.22037	25.9262	33.6837
	SF	4.189	9.72233	21.1774
	TBM	49.3573	**92.9962**	142.594
18	ABM	50.1514	97.859	141.352
	HAL	50.1514	**101.773**	**159.021**
	L	5.86185	24.7023	31.4115
	SF	4.05173	9.63763	21.0275
	TBM	**51.2912**	97.859	149.667

Table 1: Algorithm Speed (Characters Per Microsecond) in English Text Searches on Three Systems

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 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 23⟩ ≡

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

Used in part 25.

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 σ, we can use more than one element of the text sequence to compute the hash value in order to obtain σ distinct values. In the context in which the skip loop appears, we always have available at least `pattern_size` elements; whenever `pattern_size` is too small to yield σ 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 `suffix_size` be the number of sequence elements used in computing the hash function, where $1 \leq$ `suffix_size` $\leq$ `pattern_size`.

⟨Compute skip table and mismatch shift using the hash function 24⟩ ≡

```
for i in hash_range loop
  skip(i) := pattern_size - suffix_size + 1;
end loop;
for j in a + suffix_size - 1 .. m - 2 loop
  skip(hash(pattern, j)) := m - 1 - j;
end loop;
mismatch_shift := skip(hash(pattern, m - 1));
skip(hash(pattern, m - 1)) := 0;
```

Used in part 25.

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 `suffix_size` elements, where $1 \leq$ `suffix_size` $\leq$ `pattern_size`.

```
⟨Hashed Accelerated Linear algorithm 25⟩ ≡
    pattern_size := m - a; text_size := n - b; k := b;
    ⟨Handle pattern size = 1 as a special case 7a⟩
    ⟨Compute next table 59⟩
    ⟨Compute skip table and mismatch shift using the hash function 24⟩
    large := text_size + 1;
    skip(hash(pattern, m - 1)) := large;
    adjustment := large + pattern_size - 1;
    k := k - n;
    loop
      k := k + pattern_size - 1;
      exit when k >= 0;
      ⟨Scan the text using a single-test skip loop with hashing 23⟩
      ⟨Verify match or recover from mismatch 17⟩
    end loop;
    return n;
```

Used in part 56a.

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

HAL, obtained using

$$\texttt{hash}(\texttt{text}, k) = \texttt{text}(k) \text{ and (hence) } \texttt{suffix\_size} = 1,$$

By inlining `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:

$$x \equiv y \supset \texttt{hash}(x) = \texttt{hash}(y)$$

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,[2]

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

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

[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.

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.

6 Large Alphabet Case

Suppose the alphabet Σ 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 $\texttt{skip1}(x) = \texttt{skip}(x) - m$, 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.

Pattern Size	Algorithm	System 1	System 2	System 3
20	ABM	16.0893	37.3422	55.6074
	GBM	25.8827	62.3888	138.267
	HAL	13.1493	32.5853	53.8514
	HAL2	**28.7208**	**67.3143**	**146.168**
	HAL3	22.8165	63.9486	131.177
	HAL4	21.9008	58.135	113.686
	L	4.33091	16.3971	18.9477
	SF	3.28732	8.31851	16.94
	TBM	18.0395	42.6324	63.1591
50	ABM	19.5708	44.693	70.6633
	GBM	33.6343	78.046	193.67
	HAL	13.6876	33.736	60.8033
	HAL2	**49.5795**	**106.716**	275.215
	HAL3	43.4625	**106.716**	**290.505**
	HAL4	42.632	96.8349	249.004
	L	4.3519	16.6003	19.439
	SF	3.28906	8.31333	16.7599
	TBM	24.0764	54.4696	87.1514
100	ABM	21.2655	49.6781	73.4371
	GBM	36.6439	85.8841	220.311
	HAL	12.946	32.0706	56.3018
	HAL2	70.4997	163.457	389.782
	HAL3	**71.2744**	**187.673**	**460.651**
	HAL4	70.4997	168.905	**460.651**
	L	4.24474	16.0862	19.1938
	SF	3.24623	8.23929	16.5054
	TBM	27.8368	66.6732	105.566
150	ABM	24.2269	56.1366	86.667
	GBM	37.6383	86.667	205.834
	HAL	14.0205	34.5456	60.9879
	HAL2	84.3097	197.601	**548.891**
	HAL3	**91.641**	**247.001**	**548.891**
	HAL4	90.3318	235.239	494.002
	L	4.33395	16.3037	19.5258
	SF	3.28992	8.33056	17.0935
	TBM	29.1393	72.6474	107.392
200	ABM	23.9786	55.3853	86.3636
	GBM	37.0578	90.9902	212.31
	HAL	13.3106	33.3036	57.9028
	HAL2	89.3449	221.541	509.545
	HAL3	103.527	**283.081**	**636.931**
	HAL4	**105.196**	**283.081**	566.161
	L	4.26565	16.2275	19.0841
	SF	3.25946	8.28528	16.8167
	TBM	28.7321	73.8471	113.232

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

Another approach is to use HAL with, say,

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

as the hash function. In order to compare these two approaches we implemented a non-hashed version of AL using `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 decribed 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.

Pattern Size	Algorithm	System 1	System 2	System 3
2	HAL	**10.4369**	27.3645	39.5632
	L	7.11972	**28.5543**	**41.5664**
	NHAL	6.63479	12.3915	26.4818
	SF	4.48334	9.83157	23.125
4	HAL	**18.7455**	**44.375**	**64.3873**
	L	7.125	28.3082	41.5665
	NHAL	10.9539	21.0497	47.5906
	SF	4.48611	9.86112	22.8038
6	HAL	**25.3206**	**54.9243**	**86.7225**
	L	7.1179	28.4091	41.7146
	NHAL	14.2358	28.1663	63.3741
	SF	4.505	9.86663	23.0451
8	HAL	**31.1354**	**67.1919**	**94.0686**
	L	7.12946	28.6296	41.676
	NHAL	16.9606	33.5959	80.3025
	SF	4.49445	9.88709	22.7062
10	HAL	**35.7717**	**72.4895**	**112.484**
	L	7.09913	28.3655	41.2915
	NHAL	19.1634	38.3768	98.8494
	SF	4.49017	9.85507	22.8114
14	HAL	**42.9195**	**78.1701**	**149.234**
	L	7.1132	28.5491	41.0393
	NHAL	23.5262	47.5818	136.798
	SF	4.48911	9.80043	22.7996
18	HAL	**47.51862**	**96.9324**	**173.458**
	L	7.144521	28.1684	41.1963
	NHAL	26.4274	56.8225	164.785
	SF	4.48312	9.80864	22.8868

Table 3: Algorithm Speed (Characters Per Microsecond) in Large Alphabet Case on Three Systems

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, 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 genericity. 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.[3] And although BM has excellent average

[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

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.[4]

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

```
template <typename ForwardIterator1, typename ForwardIterator2>
   ForwardIterator1 search(ForwardIterator1 first1,
                           ForwardIterator1 last1,
                           ForwardIterator2 first2,
                           ForwardIterator2 last2);

template <typename ForwardIterator1, typename ForwardIterator2,
             typename BinaryPredicate>
   ForwardIterator1 search(ForwardIterator1 first1,
                           ForwardIterator1 last1,
                           ForwardIterator2 first2,
```

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.

```
                                  ForwardIterator2 last2,
                                  BinaryPredicate pred);
```

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 last1 if no such iterator is found.

Complexity: At most $(\text{last1} - \text{first1}) * (\text{last2} - \text{first2})$ 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 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]

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.

[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 broadly in other parts of the library, particularly to provide different definitions of floating point and other parameters used in numeric algorithms. The most elaborate uses of the device employ the recently added C++ feature of *partial specialization*, in which new definitions can be given with some template parameters specialized while others are left unspecialized. Few C++ compilers currently support partial specialization, but we do not need it here anyway.

⟨User level search function 37a⟩ ≡

```
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 77.

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

⟨Forward iterator case 37b⟩ ≡

```
template <typename ForwardIterator1, typename ForwardIterator2>
inline ForwardIterator1 __search(ForwardIterator1 text,
                                 ForwardIterator1 textEnd,
                                 ForwardIterator2 pattern,
                                 ForwardIterator2 patternEnd,
                                 forward_iterator_tag)
{
  return __search_L(text, textEnd, pattern, patternEnd);
}
```

```
template <typename ForwardIterator1, typename ForwardIterator2>
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) 39a⟩
  m = next.size();
  ⟨Algorithm L, optimized linear pattern search (C++) 40⟩
}
```

Used in part 77.

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`.

⟨Compute next table (C++ forward) 39a⟩ ≡

```
compute_next(pattern, patternEnd, next, pattern_iterator);
```

Used in part 37b.

⟨Define procedure to compute next table (C++ forward) 39b⟩ ≡

```
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);
      }
    }
```

Used in part 77.

Returning to the search algorithm itself, the details are as follows:

⟨Algorithm L, optimized linear pattern search (C++) 40⟩ ≡

```
    ⟨Handle pattern size = 1 as a special case (C++) 41a⟩
    p1 = pattern; ++p1;
    while (text != textEnd) {
      ⟨Scan the text for a possible match (C++) 41b⟩
      ⟨Verify whether a match is possible at the position found (C++) 42⟩
      ⟨Recover from a mismatch using the next table (C++ forward) 43⟩
    }
    return textEnd;
```

Used in part 37b.

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++) 41a⟩ ≡

```
    if (next.size() == 1)
      return find(text, textEnd, *pattern);
```

Used in parts 40, 84, 93.

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++) 41b⟩ ≡

```
    while (*text != *pattern)
      if (++text == textEnd)
        return textEnd;
```

Used in part 40.

⟨Verify whether a match is possible at the position found (C++) 42⟩ ≡

```
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 40.

⟨Recover from a mismatch using the next table (C++ forward) 43⟩ ≡

```
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 44⟩
    }
    if (text == textEnd)
      return textEnd;
  }
}
```

Used in part 40.

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

⟨Compute and return position of match 44⟩ ≡

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

Used in part 43.

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*. The code supplied includes a set of operation counting components [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, *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 [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 *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 [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 *http://www.cs.rpi.edu/~musser/gp/*. By downloading the Nuweb source file, *gensearch.w*, and using Briggs' Nuweb tool [Briggs],[6] 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

[6]We started with a version of the Nuweb tool previously modified by Ramsdell and Mengel and made additional small changes in terminology in the LaTeX file the tool produces: "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.

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 $O(mn)$ 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

[BM77] R. Boyer and S. Moore. A fast string matching algorithm. *CACM, 20(1977),762–772.* 1, 1, 3

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

[Cole96] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm, *SIAM Journal on Computing* 5 (1994): 1075–1091. 1

[CGG90] L. Colussi, Z. Galil, R. Giancarlo. On the Exact Complexity of String Matching. *Proceedings of the Thirty First Annual IEEE Symposium on the Foundations of Computer Science*, 1990, 135–143.

[DNAsource] H.S. Bilofsky, C. Burks, The GenBank(r) genetic sequence data bank. *Nucl. Acids Res.* 16 (1988), 1861–1864. 2

[Ga79] Z. Galil. On Improving the worst case running time of the Boyer-Moore string matching algorithm. *CACM* 22 (1979), 505–508.

[GS83] Z. Galil, J. Seiferas. Time space optimal string matching. *JCSS* 26 (1983), 280–294.

[DraftCPP] Accredited Standards Committee X3 (American National Standards Institute), Information Processing Systems, *Working paper for draft proposed international standard for information systems—programming language C++*. Doc No. X3J16/95-0185, WG21/N0785.[[Check for most recent version.]] 4

[GO77] L.J. Guibas, A.M. Odlyzko, A new proof of the linearity of the Boyer-Moore string searching algorithm. *Proc. 18th Ann. IEEE Symp. Foundations of Comp. Sci., 1977, 189–195*

[Horspool88] R.N. Horspool. Practical fast searching in strings *Soft.-Prac. and Exp., 10 (March 1980), 501–506* 1

[Hume88] A. Hume. A tale of two greps. *Soft.-Prac. and Exp.* 18 (November 1988), 1063–1072.

[HS91] A. Hume, S. Sunday. Fast string searching. *Soft.-Prac. and Exp.* 21 (November 1991), 1221–1248. 1, 3, 2, 5, 5

[Knuth84] D.E. Knuth, Literate programming. *Computer Journal* 27 (1984), 97–111. 1

[KMP77] D.E. Knuth, J. Morris, V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing* 6 (1977), 323–350. 1, 2

[Mu96] D.R. Musser. *Measuring Computing Times and Operation Counts*, http://www.cs.rpi.edu/musser/gp/timing.html. 5

[MS96] D.R. Musser, A. Saini. *STL Tutorial and Reference Guide: C++ Programming with Standard Template Library.* Addison-Wesley, Reading, MA, 1996. 7

[SGI96] Silicon Graphics Standard Template Library Programming Guide, online guide, http://www.sgi.com/Technology/STL/.

[Sm82] G.V. Smit. A comparison of three string matching algorithms. *Soft.-Prac. and Exp.* 12, 1 (Jan 1982), 57–66. 1

[StepanovLee] A.A. Stepanov, M. Lee, *The Standard Template Library*, Tech. Report HPL-94-34, April 1994, revised October 31, 1995. 7

[Su90] D.M. Sunday. A very fast substring search algorithm. *CACM* 33 (August 1990), 132–142. 1

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 52⟩ ≡

```
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 60, 68b, 72c.

⟨Algorithm subprogram declarations 53a⟩ ≡

> ⟨Define procedure to compute next table 58⟩
>
> ⟨Non-hashed algorithms 53b⟩
>
> ⟨Simple hash function declarations 55⟩
>
> ⟨HAL declaration 56a⟩

Used in parts 60, 68b, 72c.

⟨Non-hashed algorithms 53b⟩ ≡

```
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 59⟩
  ⟨Basic KMP 5⟩
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 59⟩
  ⟨Algorithm L, optimized linear pattern search 6⟩
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 7a⟩
  while k <= n - pattern_size loop
    ⟨Scan the text for a possible match 7b⟩
    ⟨Verify whether a match is possible at the position found 8⟩
    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 16⟩
end AL;
```

Used in part 53a.

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

⟨Simple hash function declarations 55⟩ ≡

```
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 53a.

⟨HAL declaration 56a⟩ ≡
```
function HAL(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: Integer_Sequence(hash_range);
begin
  ⟨Hashed Accelerated Linear algorithm 25⟩
end HAL;
```

Used in part 53a.

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

⟨Additional algorithms 56b⟩ ≡
```
function AL0(text, pattern: Character_Sequence;
               b, n, a, m: Integer) return Integer is
  pattern_size, j, k, d, mismatch_shift: Integer;
  next: Integer_Sequence(a .. m - 1);
  skip: Skip_Sequence(Character'Range);
begin
  ⟨Accelerated Linear algorithm, preliminary version 12⟩
end AL0;

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

function SF2(text, pattern: Character_Sequence;
             b, n, a, m: Integer) return Integer is
  pattern_size, j, k, k0, n0: Integer;
begin
  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 60, 68b, 72c.

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

⟨Define procedure to compute next table 58⟩ ≡

```
  procedure Compute_Next(pattern: Character_Sequence; a, m: Integer;
                                next: out Integer_Sequence) is
    j: Integer := a;
    t: Integer := a - 1;
  begin
    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 53a.

⟨Compute next table 59⟩ ≡

```
  Compute_Next(pattern, a, m, next);
```

Used in parts 12, 16, 25, 53b.

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" 60 ≡
     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 52⟩
        ⟨Variable declarations 61a⟩
        ⟨Algorithm subprogram declarations 53a⟩
        ⟨Additional algorithms 56b⟩
        ⟨Define procedure to read string into sequence 64⟩
        ⟨Define procedure to output sequence 65a⟩
        ⟨Define algorithm enumeration type, names, and selector function 61c⟩
        ⟨Define Report procedure 65c⟩
     begin
        ⟨Set file small.txt as input file 61b⟩
        loop
           ⟨Read test sequences from file 63a⟩
           ⟨Run tests and report results 65b⟩
        end loop;
end Test_Search;
```

⟨Variable declarations 61a⟩ ≡

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

Used in part 60.

⟨Set file small.txt as input file 61b⟩ ≡

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

Used in part 60.

⟨Define algorithm enumeration type, names, and selector function 61c⟩ ≡

```
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           ",
     "L                 ",
     "AL                ",
     "HAL               ");

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;
```

Used in parts 60, 68b, 72c.

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 63a⟩ ≡

```
    exit when Text_Io.End_Of_File;
    Get(Comment, Last);
    Put(Comment, Last); New_Line;
    ⟨Check for unexpected end of file 63b⟩

    Get(S1, Last);
    ⟨Check for unexpected end of file 63b⟩
    Put("Text sequence:    "); Put(S1, Last);
    S1_Length := Last;

    Get(S2, Last);
    Put("Pattern sequence: "); Put(S2, Last);
    S2_Length := Last;
```

Used in part 60.

⟨Check for unexpected end of file 63b⟩ ≡

```
    if Text_Io.End_Of_File then
      Put_Line("**** Unexpected end of file."); New_Line;
      raise Program_Error;
    end if;
```

Used in part 63a.

⟨Define procedure to read string into sequence 64⟩ ≡

```
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 60.

⟨Define procedure to output sequence 65a⟩ ≡

```
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 60.

⟨Run tests and report results 65b⟩ ≡

```
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 60.

⟨Define Report procedure 65c⟩ ≡

```
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 67⟩
    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 60, 68b.

⟨Output S2 67⟩ ≡

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

Used in part 65c.

Here are a few small tests.

"small.txt" 68a ≡

```
#
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.
babcbabcabcaabcabcabcacabc
abcabcacab
#
aaaaaaabcabcadefg
abcad
#
aaaaaaabcabcadefg
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" 68b ≡

```
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 52⟩
  ⟨Data declarations 70a⟩
  ⟨Algorithm subprogram declarations 53a⟩
  ⟨Additional algorithms 56b⟩
  ⟨Define algorithm enumeration type, names, and selector function 61c⟩
  ⟨Define Report procedure 65c⟩
  S2: Character_Sequence(0 .. 100);
begin
  ⟨Read test parameters 70c⟩
  ⟨Set file long.txt as input file 70b⟩
  ⟨Read character sequence from file 71a⟩
  Increment := (S1_Length - S2_Length) / Number_Of_Tests;
  ⟨Run tests searching for selected subsequences 71b⟩
end Test_Long_Search;
```

⟨Data declarations 70a⟩ ≡

```
    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 68b, 72c.

⟨Set file long.txt as input file 70b⟩ ≡

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

Used in parts 68b, 72c.

⟨Read test parameters 70c⟩ ≡

```
    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 68b, 72c.

⟨Read character sequence from file 71a⟩ ≡

```
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 68b, 72c.

⟨Run tests searching for selected subsequences 71b⟩ ≡

```
F := 0;
for K in 1 .. Number_Of_Tests loop
   ⟨Select sequence S2 to search for in S1 72a⟩
   ⟨Run tests 72b⟩
end loop;
```

Used in part 68b.

⟨Select sequence S2 to search for in S1 72a⟩ ≡
```
for I in 0 .. Pattern_Size - 1 loop
  S2(I) := S1(F + I);
end loop;
F := F + Increment;
```

Used in parts 71b, 74b.

⟨Run tests 72b⟩ ≡
```
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, 0, S1_Length, 0, S2_Length);
end loop;
New_Line;
```

Used in part 71b.

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" 72c ≡

```
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
  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 52⟩
  ⟨Algorithm subprogram declarations 53a⟩
  ⟨Additional algorithms 56b⟩
  ⟨Define algorithm enumeration type, names, and selector function 61c⟩
  ⟨Data declarations 70a⟩
  ⟨Define run procedure 74b⟩
begin
  ⟨Read test parameters 70c⟩
  ⟨Set file long.txt as input file 70b⟩
  ⟨Read character sequence from file 71a⟩
  Increment := (S1_Length - S2_Length) / Number_Of_Tests;
  Base_Time := 0.0;
  ⟨Run and time tests searching for selected subsequences 74a⟩
end Time_Long_Search;
```

⟨Run and time tests searching for selected subsequences 74a⟩ ≡

```
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 72c.

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 74b⟩ ≡

```
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);
begin
  F := 0;
```

```
    Total_Search := 0;
    Start_Time := Clock;
    for I in 1 .. Number_Of_Tests loop
      ⟨Select sequence S2 to search for in S1 72a⟩
      P := Algorithm(K, S1, S2, 0, Text_Size, 0, Pattern_Size);
      Total_Search := Total_Search + P + Pattern_Size;
    end loop;
    Finish_Time := Clock;
    ⟨Output statistics 75⟩
  end Run;
```

Used in part 72c.

⟨Output statistics 75⟩ ≡

```
  Time_Taken := Long_Float((Finish_Time - Start_Time) / Milliseconds(1))
                  / 1000.0 - Base_Time;
  Put("Total search length: ");
  Put(Total_Search); Put(" bytes."); New_Line;
  Put("Time:  "); My_Float.Put(Time_Taken, 5, 4, 0);
  Put(" seconds."); New_Line;
  if K /= Dummy then
    Put("Speed: ");
    My_Float.Put(Long_Float(Total_Search) / 1_000_000.0 / Time_Taken, 5, 2, 0);
    Put(" MBytes/second."); New_Line;
  else
```

```
    Base_Time := Time_Taken;
  end if;
  New_Line;
```

Used in part 74b.

B 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 `#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" 77 ≡
     #ifndef NEW_SEARCH
     #  define NEW_SEARCH
     #  include <vector>
     #  include "search_traits.h"
     #  include <iterator>
     using namespace std;

     ⟨Define procedure to compute next table (C++) 88b⟩
     ⟨Define procedure to compute next table (C++ forward) 39b⟩
     ⟨User level search function 37a⟩
     ⟨Forward iterator case 37b⟩
     ⟨Bidirectional iterator case 81⟩
     ⟨HAL with random access iterators, no trait passed 82⟩
     ⟨User level search function with trait argument 83⟩
     #endif
```

B.1.2 Search Traits

"search_traits.h" 78a ≡
```
#ifndef SEARCH_HASH_TRAITS
#  define SEARCH_HASH_TRAITS
⟨Generic search trait 78b⟩
⟨Search traits for character sequences 80⟩
#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 78b⟩ ≡
```
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 78a.

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 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:

⟨Search traits for character sequences 80⟩ ≡
```
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;
  }
};
```

Used in part 78a.

B.1.3 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 version.

⟨Bidirectional iterator case 81⟩ ≡
```
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);
}
```

Used in part 77.

When we have random access iterators and no search trait is passed as an 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 resort to algorithm L.

⟨HAL with random access iterators, no trait passed 82⟩ ≡

```
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));
}
```

Used in part 77.

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

⟨User level search function with trait argument 83⟩ ≡

```
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++) 84⟩
}
```

Used in part 77.

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++) 84⟩ ≡
     k = 0;
     text_size = textEnd - text;
     ⟨Compute next table (C++) 89⟩
     ⟨Handle pattern size = 1 as a special case (C++) 41a⟩
     ⟨Compute skip table and mismatch shift using the hash function (C++) 88a⟩
     large = text_size + 1;
     adjustment = large + pattern_size - 1;
     skip[Trait::hash(pattern + pattern_size - 1)] = large;
     k -= text_size;
     for(;;) {
       k += pattern_size - 1;
       if (k >= 0) break;
       ⟨Scan the text using a single-test skip loop with hashing (C++) 85a⟩
       ⟨Verify match or recover from mismatch (C++) 85b⟩
     }
     return textEnd;
```

Used in part 83.

⟨Scan the text using a single-test skip loop with hashing (C++) 85a⟩ ≡

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

Used in part 84.

⟨Verify match or recover from mismatch (C++) 85b⟩ ≡

```
if (textEnd[k] != pattern[0])
  k += mismatch_shift;
else {
  ⟨Verify the match for positions 1 through pattern_size - 1 (C++) 86⟩
  if (mismatch_shift > j)
    k += mismatch_shift - j;
  else
    ⟨Recover from a mismatch using the next table (C++) 87⟩
}
```

Used in parts 84, 93.

⟨Verify the match for positions 1 through pattern_size - 1 (C++) 86⟩ ≡

```
j = 1;
for (;;) {
  ++k;
  if (textEnd[k] != pattern[j])
    break;
  ++j;
  if (j == pattern_size)
    return textEnd + k - pattern_size + 1;
}
```

Used in part 85b.

⟨Recover from a mismatch using the next table (C++) 87⟩ ≡

```
for (;;) {
  j = next[j];
  if (j < 0) {
    ++k;
    break;
  }
  if (j == 0)
    break;
  while (textEnd[k] == pattern[j]) {
    ++k; ++j;
    if (j == pattern_size) {
      return textEnd + k - pattern_size;
    }
    if (k == 0)
      return textEnd;
  }
}
```

Used in part 85b.

B.1.4 Skip Table Computation

⟨Compute skip table and mismatch shift using the hash function (C++) 88a⟩ ≡

```
m = next.size();
for (i = 0; i < Trait::hash_range_max; ++i)
  skip.push_back(m - Trait::suffix_size + 1);
for (j = Trait::suffix_size - 1; j < m - 1; ++j)
  skip[Trait::hash(pattern + j)] = m - 1 - j;
mismatch_shift = skip[Trait::hash(pattern + m - 1)];
skip[Trait::hash(pattern + m - 1)] = 0;
```

Used in part 84.

B.1.5 Next Table Procedure and Call

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

⟨Define procedure to compute next table (C++) 88b⟩ ≡

```
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);
    }
  }
```

Used in part 77.

⟨Compute next table (C++) 89⟩ ≡

```
  compute_next(pattern, patternEnd, next);
```

Used in parts 84, 93.

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" 90a ≡
⟨Experimental search function with skip loop without hashing 90b⟩

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

⟨Experimental search function with skip loop without hashing 90b⟩ ≡
```
#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};
};

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++) 93⟩
}
```

Used in part 90a.

⟨Accelerated Linear algorithm, no hashing (C++) 93⟩ ≡

```
k = 0;
text_size = textEnd - text;
⟨Compute next table (C++) 89⟩
⟨Handle pattern size = 1 as a special case (C++) 41a⟩
⟨Compute skip table and mismatch shift, no hashing (C++) 94a⟩
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++) 94b⟩
  ⟨Verify match or recover from mismatch (C++) 85b⟩
}
return textEnd;
```

Used in part 90b.

⟨Compute skip table and mismatch shift, no hashing (C++) 94a⟩ ≡

```
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 93.

⟨Scan the text using a single-test skip loop, no hashing (C++) 94b⟩ ≡

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

Used in part 93.

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" 94c ≡

⟨Define DNA search traits 96⟩

```
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 patternEnd)
{
  return search_hashed(text, textEnd, pattern, patternEnd,
                       static_cast<search_trait_dna5*>(0));
}
```

⟨Define DNA search traits 96⟩ ≡

```
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 94c.

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.

⟨Include algorithms header with existing search function renamed 98⟩ ≡

```
#define search stl_search
#define __search __stl_search
#include <algorithm>
#undef search
#undef __search
```

Used in parts 99, 105, 111b, 117, 121, 128, 132b.

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" 99 ≡
    ⟨Include algorithms header with existing search function renamed 98⟩
    #include <iostream>
    #include <fstream>
    #include "new_search.h"
    #include "hume.hh"
    #include "DNA_search.h"
    using namespace std;
    int Base_Line;
    ⟨Define procedure to read string into sequence (C++) 103b⟩
    typedef unsigned char data;
    ⟨Define algorithm enumeration type, names, and selector function (C++) 100⟩
    ⟨Define Report procedure (C++) 104b⟩
    int main()
    {
      ostream_iterator<char> out(cout, "");
      ifstream ifs("small.txt");
      vector<data> Comment, S1, S2;
      const char* separator = "";
      for (;;) {
        ⟨Read test sequences from file (C++) 102⟩
        ⟨Run tests and report results (C++) 104a⟩
      }
      return 0;}
```

⟨Define algorithm enumeration type, names, and selector function (C++) 100⟩ ≡

```
enum algorithm_enumeration {
     Dummy, SF, L, HAL, ABM, TBM, GBM, HAL2, HAL3, HAL4, HAL5
};
const char* algorithm_names[] = {
     "selection code", "SF", "L", "HAL", "ABM", "TBM", "GBM",
     "HAL2", "HAL3", "HAL4", "HAL5"
};

#ifndef DNA_TEST
  algorithm_enumeration alg[] = {Dummy, SF, L, HAL, ABM, TBM};
  const char textFileName[] = "long.txt";
  const char wordFileName[] = "words.txt";
#else
  algorithm_enumeration alg[] = {Dummy, SF, L, HAL, ABM, GBM,
                                 HAL2, HAL3, HAL4, HAL5};
  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 99, 105, 111b.

⟨Read test sequences from file (C++) 102⟩ ≡

```
    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++) 103a⟩
    cout << "Text string:......";
    copy(S1.begin(), S1.end(), out);
    cout << endl;

    get(ifs, S2);
    ⟨Check for unexpected end of file (C++) 103a⟩
    cout << "Pattern string:...";
    copy(S2.begin(), S2.end(), out); cout << endl;
```

Used in part 99.

⟨Check for unexpected end of file (C++) 103a⟩ ≡

```
if (ifs.eof()) {
  cout << "**** Unexpected end of file." << endl;
  exit(1);
}
```

Used in part 102.

⟨Define procedure to read string into sequence (C++) 103b⟩ ≡

```
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 99.

⟨Run tests and report results (C++) 104a⟩ ≡

```
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 99, 110b, 111a.

⟨Define Report procedure (C++) 104b⟩ ≡

```
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;
  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;
      }
```

Used in parts 99, 105, 128.

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" 105 ≡

```
⟨Include algorithms header with existing search function renamed 98⟩
#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;
⟨Define algorithm enumeration type, names, and selector function (C++) 100⟩
⟨Define Report procedure (C++) 104b⟩
int main()
{
  unsigned int F, K, j;
  ⟨Read test parameters (C++) 107⟩
  ⟨Read dictionary from file, placing words of size j in dictionary[j] 108⟩
  ⟨Read character sequence from file (C++) 109b⟩
  for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
    ⟨Trim dictionary[Pattern_Size[j]] to have at most Number_Of_Tests words 109a⟩
    Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
    cerr << Pattern_Size[j] << " " << flush;
    const char* separator = "";
    ⟨Output header (C++) 110a⟩
    ⟨Run tests searching for selected subsequences (C++) 110b⟩
    ⟨Run tests searching for dictionary words (C++) 111a⟩
  }
}
```

⟨Read test parameters (C++) 107⟩ ≡

```
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;
```

Used in parts 105, 111b, 117, 121, 128, 132b.

⟨Read dictionary from file, placing words of size j in dictionary[j] 108⟩ ≡

```
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);
}
```

Used in parts 105, 111b, 121.

⟨Trim dictionary[Pattern_Size[j]] to have at most Number_Of_Tests words 109a⟩ ≡

```
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 105, 111b, 121.

⟨Read character sequence from file (C++) 109b⟩ ≡

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

Used in parts 105, 111b, 121.

⟨Output header (C++) 110a⟩ ≡

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

Used in parts 105, 111b, 117, 121, 128, 132b.

⟨Run tests searching for selected subsequences (C++) 110b⟩ ≡

```
F = 0;
for (K = 1; K <= Number_Of_Tests; ++K) {
   ⟨Select sequence S2 to search for in S1 (C++) 110c⟩
   ⟨Run tests and report results (C++) 104a⟩
}
```

Used in parts 105, 128.

⟨Select sequence S2 to search for in S1 (C++) 110c⟩ ≡

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

Used in part 110b.

⟨Run tests searching for dictionary words (C++) 111a⟩ ≡

```
for (K = 0; K < dictionary[Pattern_Size[j]].size(); ++K) {
  S2 = dictionary[Pattern_Size[j]][K];
  ⟨Run tests and report results (C++) 104a⟩
}
```

Used in part 105.

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" 111b ≡

```
⟨Include algorithms header with existing search function renamed 98⟩
#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++) 100⟩
⟨Define run procedure (C++ forward) 113⟩

int main()
{
  int j;
  ⟨Read test parameters (C++) 107⟩
  ⟨Read character sequence from file (C++) 109b⟩
  ⟨Read dictionary from file, placing words of size j in dictionary[j] 108⟩
  for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
    ⟨Trim dictionary[Pattern_Size[j]] to have at most Number_Of_Tests words 109a⟩
    Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
    ⟨Output header (C++) 110a⟩
    cerr << Pattern_Size[j] << " " << flush;
    ⟨Run and time tests searching for selected subsequences (C++) 115a⟩
  }
  cerr << endl;
}
```

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) 113⟩ ≡

```
  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 114⟩
    }
    for (K = 0; K < dictionary.size(); ++K) {
      S2 = dictionary[K];
      ⟨Run algorithm and record search distance 114⟩
    }
    Finish_Time = clock();
    ⟨Output statistics (C++) 115b⟩
  }
```

Used in parts 111b, 117, 132b.

⟨Run algorithm and record search distance 114⟩ ≡

```
  Algorithm(k, S1, S2, P);
  d = 0;
  distance(S1.begin(), P, d);
  Total_Search += d + Pattern_Size;
```

Used in parts 113, 125.

⟨Run and time tests searching for selected subsequences (C++) 115a⟩ ≡

```
Base_Time = 0.0;
for (int k = 0; k < number_of_algorithms; ++k) {
  if (k != 0)
    cout << "Timing " << algorithm_names[k] << ":" << endl;
  Run(k, S1, dictionary[Pattern_Size[j]], Pattern_Size[j]);
}
cout << endl;
```

Used in parts 111b, 117, 121, 132b.

⟨Output statistics (C++) 115b⟩ ≡

```
Time_Taken = (Finish_Time - Start_Time)/CLOCKS_PER_SEC - Base_Time;
if (k == 0)
  Base_Time = Time_Taken;
else {
  cout << "Total search length: " << Total_Search << " elements" << endl;
  cout << "Time: " << Time_Taken << " seconds." << endl;
  double Speed = Time_Taken == 0.0 ? 0.0 :
    static_cast<double>(Total_Search) / 1000000 / Time_Taken;
  cout << "Speed: " << Speed << " elements/microsecond." << endl << endl;
}
```

Used in part 113.

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) 116⟩ ≡

```
enum algorithm_enumeration {
     Dummy, SF, L, HAL, NHAL
};
const char* algorithm_names[] = {
     "selection code", "SF", "L", "HAL", "NHAL"
};

const int number_of_algorithms = 5;

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:
     // 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 NHAL:
       result = search_no_hashing(x.begin(), x.end(), y.begin(), y.end() ); return;
    }
    result = x.begin(); return;
  }
```

Used in part 117.

"experimental_search.cpp" 117 ≡

```
⟨Include algorithms header with existing search function renamed 98⟩
#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;
⟨Define algorithm enumeration type, names, and selector function (C++ large alphabet) 116⟩
⟨Define run procedure (C++ forward) 113⟩
⟨Define RandomNumberGenerator class 119a⟩

int main()
{
  int j;
  ⟨Read test parameters (C++) 107⟩
  ⟨Generate data sequence 119b⟩
  ⟨Generate dictionary 120⟩
  for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
    Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
    ⟨Output header (C++) 110a⟩
    cerr << Pattern_Size[j] << " " << flush;
    ⟨Run and time tests searching for selected subsequences (C++) 115a⟩
  }
  cerr << endl;
}
```

⟨Define RandomNumberGenerator class 119a⟩ ≡

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

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

Used in part 117.

⟨Generate data sequence 119b⟩ ≡

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

Used in part 117.

⟨Generate dictionary 120⟩ ≡

```
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 117.

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" 121 ≡
    ⟨Include algorithms header with existing search function renamed 98⟩
    #include "new_search.h"
    #include "hume.hh"
    #include <iterator>
    #include <vector>
    #include <map>
    #include <iostream>
    #include <fstream>
    #include <ctime>
    #include <string>
    using namespace std;

    ⟨Define types needed for counting operations 122⟩
    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++ counter) 123⟩
    recorder<> stats[number_of_algorithms];
    ⟨Define run procedure (C++ counter) 125⟩

    int main()
    {
```

```
    int j;
    ⟨Read test parameters (C++) 107⟩
    ⟨Read character sequence from file (C++) 109b⟩
    ⟨Read dictionary from file, placing words of size j in dictionary[j] 108⟩
    for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
      ⟨Trim dictionary[Pattern_Size[j]] to have at most Number_Of_Tests words 109a⟩
      Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
      ⟨Output header (C++) 110a⟩
      cerr << Pattern_Size[j] << " " << flush;
      ⟨Run and time tests searching for selected subsequences (C++) 115a⟩
    }
    cerr << endl;
  }
```

⟨Define types needed for counting operations 122⟩ ≡

```
  #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();}
};
```

Used in part 121.

⟨Define algorithm enumeration type, names, and selector function (C++ counter) 123⟩ ≡

```
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 121.

⟨Define run procedure (C++ counter) 125⟩ ≡

```
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 114⟩
  }
  for (K = 0; K < dictionary.size(); ++K) {
    S2 = dictionary[K];
    ⟨Run algorithm and record search distance 114⟩
  }
  stats[k].record();
  Finish_Time = clock();
  ⟨Output statistics (C++ counter) 127⟩
}
```

Used in part 121.

⟨Output statistics (C++ counter) 127⟩ ≡

```
Time_Taken = (Finish_Time - Start_Time)/CLOCKS_PER_SEC - Base_Time;
if (k == 0)
  Base_Time = Time_Taken;
else {
//  data::report(cout, Total_Search, 4);
//  citer::report(cout, Total_Search, 4);
//  cdistance::report(cout, Total_Search, 4);
  stats[k].hide_all();
  stats[k].show(recorder<>::ASSIGNMENT, true);
  stats[k].show(recorder<>::LESS_THAN, true);

  cout << "Total search length: " << Total_Search << " elements" << endl;
  cout << "Time: " << Time_Taken << " seconds." << endl;
  double Speed = Time_Taken == 0.0 ? 0.0 :
    (double)Total_Search / 1000000 / Time_Taken;
  cout << "Speed: " << Speed << " elements/microsecond." << endl << endl;
}
```

Used in part 125.

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" 128 ≡

```
⟨Include algorithms header with existing search function renamed 98⟩
#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;
⟨Define search trait for word searches 130a⟩
⟨Define algorithm enumeration type, names, and selector function (C++ word) 130b⟩
⟨Define Report procedure (C++) 104b⟩
int main()
{
  int F, K, j;
  ⟨Read test parameters (C++) 107⟩
  typedef map<int, vector<sequence >, less<int> > map_type;
  map_type dictionary;
  ⟨Read word sequence from file (C++) 132a⟩
  cout << S1.size() << " words read." << endl;
  const char* separator = " ";
  for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
    Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
    ⟨Output header (C++) 110a⟩
    ⟨Run tests searching for selected subsequences (C++) 110b⟩
  }
}
```

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 130a⟩ ≡

```
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];
  }
};
```

Used in parts 128, 132b.

⟨Define algorithm enumeration type, names, and selector function (C++ word) 130b⟩ ≡

```
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;
}
```

Used in parts 128, 132b.

⟨Read word sequence from file (C++) 132a⟩ ≡

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

Used in parts 128, 132b.

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" 132b ≡

```
⟨Include algorithms header with existing search function renamed 98⟩
#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;
⟨Define search trait for word searches 130a⟩
⟨Define algorithm enumeration type, names, and selector function (C++ word) 130b⟩
⟨Define run procedure (C++ forward) 113⟩
int main()
{
  int j;
  ⟨Read test parameters (C++) 107⟩
  typedef map<int, vector<sequence >, less<int> > map_type;
  map_type dictionary;
  ⟨Read word sequence from file (C++) 132a⟩
  cout << S1.size() << " words read." << endl;
  for (j = 0; j < Number_Of_Pattern_Sizes; ++j) {
    Increment = (S1.size() - Pattern_Size[j]) / Number_Of_Tests;
    ⟨Output header (C++) 110a⟩
    ⟨Run and time tests searching for selected subsequences (C++) 115a⟩
  }
}
```

C Index of Part Names

Pattern Size	Algorithm	Comparisons	Other Accesses	Big Jumps	Other Iter Ops	Distance Ops	Total Ops
2	SF	1.036	0.001	0.000	4.192	2.002	7.231
	L	1.028	0.001	0.000	4.095	0.177	5.301
	HAL	0.018	0.513	0.551	1.104	2.431	4.617
	ABM	0.017	0.528	—	—	—	—
	TBM	0.021	0.511	—	—	—	—
4	SF	1.034	0.000	0.000	4.170	2.000	7.203
	L	1.031	0.000	0.000	4.098	0.159	5.288
	HAL	0.013	0.266	0.291	0.583	0.658	1.811
	ABM	0.013	0.277	—	—	—	—
	TBM	0.014	0.266	—	—	—	—
6	SF	1.042	0.000	0.000	4.211	2.000	7.254
	L	1.037	0.000	0.000	4.119	0.194	5.350
	HAL	0.011	0.189	0.211	0.422	0.482	1.315
	ABM	0.012	0.198	—	—	—	—
	TBM	0.012	0.189	—	—	—	—
8	SF	1.048	0.000	0.000	4.243	2.000	7.291
	L	1.042	0.000	0.000	4.135	0.220	5.396
	HAL	0.010	0.150	0.170	0.339	0.392	1.060
	ABM	0.011	0.157	—	—	—	—
	TBM	0.011	0.150	—	—	—	—
10	SF	1.052	0.000	0.000	4.263	2.000	7.315
	L	1.044	0.000	0.000	4.142	0.233	5.418
	HAL	0.009	0.126	0.144	0.289	0.337	0.905
	ABM	0.010	0.132	—	—	—	—
	TBM	0.010	0.126	—	—	—	—
14	SF	1.077	0.000	0.000	4.384	2.000	7.460
	L	1.060	0.000	0.000	4.197	0.328	5.585
	HAL	0.010	0.105	0.125	0.250	0.305	0.796
	ABM	0.010	0.109	—	—	—	—
	TBM	0.011	0.105	—	—	—	—
18	SF	1.105	0.000	0.000	4.525	2.000	7.629
	L	1.077	0.000	0.000	4.257	0.436	5.770
	HAL	0.011	0.096	0.117	0.234	0.295	0.753
	ABM	0.010	0.099	—	—	—	—
	TBM	0.011	0.096	—	—	—	—

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