The performance of single-keyword and multiple-keyword pattern matching algorithms

Bruce W. Watson
Faculty of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB
Eindhoven, The Netherlands
email: watson@win.tue.nl

May 11, 1994

Abstract

This paper presents a toolkit of pattern matching algorithms, performance data on the algorithms, and recommendations for the selection of an algorithm (given a particular application). The pattern matching problem is: given a finite non-empty set of keywords and an input string, find all occurrences of any of the keywords in the input string.

The pattern matching toolkit (written in the C programming language, and freely available) contains implementations of the Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick, and Commentz-Walter algorithms. The algorithms are implemented directly from the abstract algorithms derived and presented in the taxonomy of Watson and Zwaan [WZ92]. The toolkit provides one of the few known correct implementations of the Commentz-Walter precomputation algorithm.

The performance of all of the algorithms (running on a variety of workstation hardware) was measured on two types of input: English text and genetic sequences. The input data, which is the same as that used in the benchmarks of Hume and Sunday [HS91], were chosen to be representative of two of the typical uses of pattern matching algorithms. The differences between natural language text and genetic sequences serve to highlight the strengths and weaknesses of each of the algorithms. Until now, the performance of the multiple-keyword algorithms (Aho-Corasick and Commentz-Walter) had not been extensively measured.

The Knuth-Morris-Pratt and Aho-Corasick algorithms performed linearly and consistently (on widely varying keyword sets), as their theoretical running time predicts. The Commentz-Walter algorithm (and its variants) displayed more interesting behaviour, greatly out-performing even the best Aho-Corasick variant on a large portion of the input data. The recommendations section of this paper details the conditions under which a particular algorithm should be chosen.

Contents

List of Figures

1 Introduction

This paper presents a toolkit of keyword pattern matching algorithms, performance data on the algorithms, and recommendations for the selection of an algorithm (given a specific application area).

The keyword pattern matching problem is: given a finite non-empty set of keywords and an input string, find all occurrences of any of the keywords in the input string. It is assumed that the keyword set will remain relatively unchanged, while various different input strings may be used. This means that the keyword set can be used in some precomputation, while precomputation involving the input string is undesirable. Since the time involved in pattern matching usually far outweighs the time involved in precomputation, the performance of the precomputation algorithms is not discussed in this paper.

The problem, and algorithms solving it, are discussed in detail in the taxonomy of pattern matching algorithms appearing in [WZ92]. The taxonomy concentrates on the systematic derivation (with correctness arguments) of several pattern matching algorithms and their associated precomputation algorithms. Given the algorithm derivations appearing in the taxonomy, the C versions of the algorithms were easily developed. The algorithms derived in the taxonomy include:

- The Knuth-Morris-Pratt (KMP) and twelve variants of the Boyer-Moore (BM) algorithms. These are single-keyword algorithms (they require that the keyword set is a singleton set).
- Two variants of the Aho-Corasick (AC) and two variants of the Commentz-Walter (CW) algorithms (one of which is sometimes called the multiple-keyword Boyer-Moore algorithm). These are multiple-keyword algorithms.

In this report, we address the practical aspects (performance and implementation) of these algorithms. The algorithms have been implemented in the C programming language, and are available as a pattern matching toolkit (see Appendix A for a description of the toolkit and how to obtain it). While the reader may find the C programs difficult to understand in isolation, the abstract versions of the algorithms (presented in the taxonomy [WZ92]) can make the C versions easier to understand.

All of the algorithms considered in this paper have worst-case running time linear in the length of the input string. The running time of the optimized AC algorithm is independent of the keyword set, while that of the KMP, failure-function AC, BM, and CW algorithms depends (linearly in the case of BM and CW) upon the length of the longest keyword in the keyword set. The KMP, failure-function AC, BM, and CW algorithms can be expected to depend slightly on the keyword set size. Unfortunately, little is known about the relative performance (in practice) of the multiple-keyword algorithms. Only the Aho-Corasick algorithms are used extensively. The Commentz-Walter algorithms are used rarely (if ever), due to the difficulty in correctly deriving the precomputation algorithms for the CW algorithms.

The performance of the single-keyword algorithms in practice has been studied:

- In [Smit82], Smit compares the theoretical running time and the practical running time of the Knuth-Morris-Pratt algorithm, a rudimentary version of the Boyer-Moore algorithm, and a brute-force algorithm.
- In [HS91], Hume and Sunday constructed a taxonomy and explored the performance of most existing versions of the single-keyword Boyer-Moore pattern matching algorithm. Their extensive testing singled out several particularly efficient versions for use in practical applications.
- In [Pirk92], Pirklbauer compares several versions of the Knuth-Morris-Pratt algorithm, several versions of the Boyer-Moore algorithm, and a brute-force algorithm. Since Pirklbauer did not construct a taxonomy of the algorithms, the algorithms are somewhat difficult to compare to one another and the testing of the Boyer-Moore variants is not quite as extensive as the Hume and Sunday taxonomy.

In this paper, we adopt the approach (due to Hume and Sunday) of evaluating the algorithms on two types of input data: natural language input strings, and input strings encoding genetic (DNA) information. In order to compare our test results with those of Hume and Sunday, we use a superset of the test data they used in [HS91].

This paper is structured as follows:

- Section 2 briefly outlines the algorithms tested.
- Section 3 describes the testing methodology, including the test environment, test data (and related statistics), and testing problems that were encountered.
- Section 4 presents the results of the testing. Most of the results are presented in the form of performance graphs.
- Section 5 gives the conclusions of this paper.
- Section 6 gives the recommendations of this paper.
- Appendix A explains and presents all of the pattern matching and precomputation algorithms (implemented in C) used in the benchmarking.

Acknowledgements: I would like to acknowledge the assistance of the following people: Pieter 't Hoen (for assisting in the benchmarking), Carel Braam (for making fast hardware available for the benchmarking), and Kees Hemerik, Nanette Saes, Tom Verhoeff, and Gerard Zwaan (for providing technical and style comments on this paper).

2 The algorithms

The algorithms compared in this paper are:

- The Knuth-Morris-Pratt algorithm (KMP). This algorithm combines the use of indexing into the input string (and the single-keyword pattern) with a precomputed "failure-function" to simulate a finite automaton. The algorithm never backtracks in the input string.

- The optimized Aho-Corasick algorithm (AC-OPT). This algorithm uses a form of finite automaton (known as a Moore machine [HU79]) to find matches. The Moore machine detects all matches ending at any given character of the input string. The algorithm never backtracks in the input string and examines each character of the input string only once.

- The failure-function Aho-Corasick algorithm (AC-FAIL). This algorithm is similar to the AC-OPT algorithm. The Moore machine used in AC-OPT is compressed into two data-structures: a forward trie [Fre60], and a failure-function. These two data-structures can be stored more space-efficiently than the full Moore machine[1], with a penalty to the running time of the algorithm. The algorithm never backtracks in the input string, but it may examine a single character more than once before proceeding; despite this, it is still linear in the length of the input string.

- The Commentz-Walter algorithms (CW). In all versions of the CW algorithms, a common program skeleton is used with different shift functions. The CW algorithms are similar to the Boyer-Moore algorithm. A match is attempted by scanning backwards through the input string. At the point of a mismatch, something is known about the input string (by the number of characters that were matched before the mismatch). This information is then used as an index into a precomputed table to determine a distance by which to shift before commencing the next match attempt. The details and precomputation of these algorithms can be found in [WZ92]. The two different shift functions compared in this paper are: the multiple-keyword Boyer-Moore shift function (CW-BM) and the Commentz-Walter normal shift function (CW-NORM). In [WZ92], it is shown that the CW-NORM shift function always yields a shift that is at least as great as that yielded by the CW-BM shift function.

The algorithms (presented in Appendix A) are implemented in the C programming language, as defined by the International Standards Organization. They are also available electronically by anonymous ftp (as outlined in Appendix A). The C language was used in order to extract the maximum performance from the implementations — on most workstations, the C compiler is the one with the highest quality of optimization.

Efforts were made to implement the algorithms as efficiently as possible, while preserving readability. For example, in the algorithms used in the performance tests, indexing (as opposed to pointers) was used when accessing characters of the input string; most optimizing compilers are able to "pointerize" such indices. (The pattern matching toolkit presented in Appendix A contains pointer versions of the algorithms which can be used with non-optimizing compilers.)

The performance of the precomputation algorithms (also appearing in Appendix A) was not extensively measured. Some simple measurements, however, indicate that all of the algorithms required similar precomputation times for a given set of keywords.

[1]In [WZ92], it is shown that the Moore machine corresponding to a pattern set P is the minimal Moore that can perform the same pattern matching job.

3 Testing methodology

The performance of each of the pattern matching algorithms is linear in the size of the input string. The performance of the AC variants and the KMP algorithm is largely independent of the keyword set, while the CW algorithms running time depends on the keyword set. The testing of these algorithms is intended to determine the relative performance of the algorithms on two types of test data (each having different characteristics):

- English text was chosen as the first type of input data since it is the most common input to pattern matching programs such as `fgrep`, and
- DNA sequences were chosen as the second type of input data since genome mapping projects make heavy use of pattern matching algorithms, and the characteristics of the input data are unlike the natural language input data.

The testing of the algorithms is also intended to explore the dependence of the performance of the Commentz-Walter algorithms upon the keyword sets.

3.1 Test environment

The tests were performed on a Digital Equipment Corporation Alpha workstation (running OSF/1) with a 100 Mhz clock. A smaller number of tests were also performed on a Hewlett-Packard Snake workstation and a Sun SPARC Station 1+. The tests showed that the relative performance data gathered on the Alpha is typical of what could be found on other high performance workstations.

During all tests, only one user (the tester) was logged-in. Typical UNIX background processes ran during the tests. Since the running time of the algorithm was obtained using the `getrusage` system call, these background processes did not skew the algorithm performance data. The data-structures used in the testing were all in physical memory during the tests. No page faults were reported by `getrusage`, and all data was accessed before the timed run (to ensure that they were in physical memory). Methods of disabling the cache memory were not explored. All of the frequently accessed data-structures were too large (frequently a megabyte) to fit in a first level cache. The linear memory access behaviour of all of the algorithms means that performance skewing due to caching effects was negligible.

3.2 Natural language test data

The test data is a superset of that used by Hume and Sunday [HS91]. The input alphabet consists of the 52 upper-case and lower-case letters of the alphabet, the space, and the new-line characters. The input string is a large portion (999952 bytes) of the bible, organized as one word per line. Each algorithm was run 30 times over the input string, effectively giving an input string of approximately 28 Megabytes in length. The bible was chosen as input since Hume and Sunday used it (and we wish to facilitate comparison of our data with that of Hume and Sunday), and it is freely redistributable.

Some data on the words making up the input string is shown in the following table:

Word length	*Number of words*
1	4403
2	32540
3	55212
4	43838
5	23928
6	13010
7	9946
8	6200
9	4152
10	1851
11	969
12	407
13	213
14	83
15	22
16	5
17	1

There are a total of 196780 words; the mean word length is 4.08 and the standard deviation is 1.97.

The single-keywords sets are the same as those used by Hume and Sunday. They consist of 500 randomly chosen words, 428 of which appear in the input string. Some data on the keywords are shown in the following table:

Word length	*Number of words*
1	0
2	1
3	14
4	47
5	68
6	103
7	79
8	79
9	49
10	29
11	14
12	9
13	7
14	0
15	0
16	1

Note that there are no words of length 1, 14, or 15. The mean word length is 6.95 and the standard deviation is 2.17.

The multiple-keyword sets were all subsets of the words appearing in the input string. Preliminary testing showed that the performance of the algorithms (on English text) is almost entirely independent of the number of matches in the input string. A total of 4174 different keyword sets were generated using the random number generator appearing in [PTVF92]. The relatively even distribution of keyword set sizes can be seen in the following table:

Keyword set size	*Number of sets*
1	212
2	217
3	206
4	193
5	199
6	200
7	217
8	206
9	234
10	202
11	235
12	219
13	210
14	210
15	177
16	221
17	196
18	222
19	200
20	198

The mean keyword set size is 10.47 and the standard deviation is 5.73.

An additional statistic (concerning the multiple-keyword sets) was recorded: the length of the shortest keyword in any given keyword set; for a given keyword set the BM and CW shift distances are bounded above by the length of the shortest keyword in the set. The data are as follows:

Length of shortest keyword	*Number of keyword sets*
1	4
2	196
3	1033
4	1698
5	719
6	286
7	100
8	70
9	47
10	12
11	5
12	2
13	1
14	1

The mean is 4.19 and the standard deviation is 1.36.

3.3 DNA sequence test data

The test data consists of the input string used by Hume and Sunday [HS91], and randomly generated keyword sets. The input alphabet consists of the four letters a, c, g, and t used to encode DNA, and the new-line character. The input string is a portion (997642 bytes) of the GenBank DNA database, as distributed by Hume and Sunday. Each algorithm was run 30 times over the input string, effectively giving an input string of approximately 28 Megabytes in length.

A total of 450 keyword sets were randomly chosen (the keywords are all substrings of the input string). Within each keyword set, all keywords were of the same length. The keyword sets were distributed evenly with set sizes ranging from 1 to 10 and keyword lengths ranging from 100 to 900 (in increments of 100).

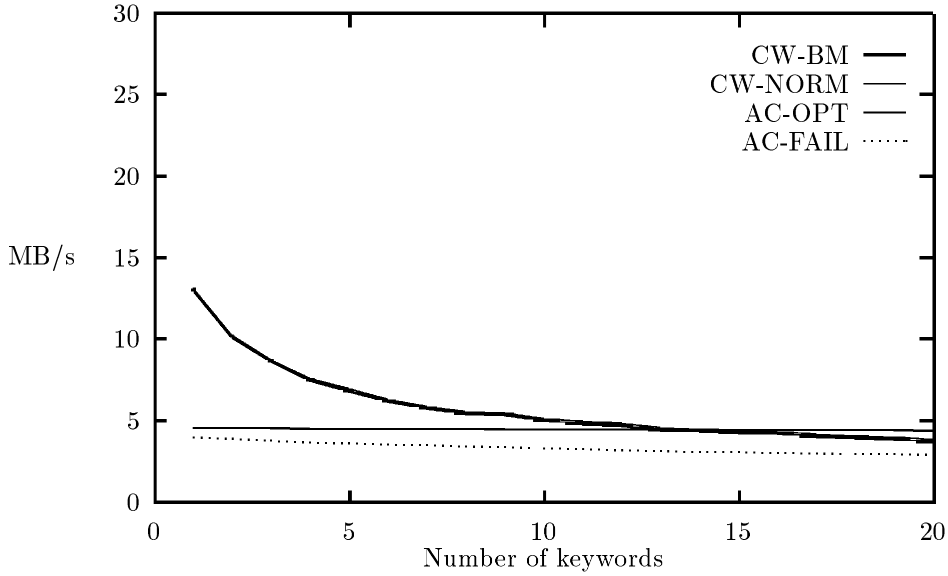

Figure 1: Algorithm performance (in megabytes/second) versus keyword set size. The performance of the CW-BM and CW-NORM algorithms are superimposed (shown as the solid descending line).

4 Results

The performance of the algorithms was measured on thirty iterations over the input string. The running time on both types of test data was found to be independent of the number of matches found in the input string. This is mostly due to the fact that the algorithms register a match by recording one integer, and incrementing a pointer — cheap operations on most processors.

The performance of each algorithm was graphed against the size of the keyword sets, and against the lengths of the shortest keyword in a set. Graphing the performance against other statistics (such as the sum of the lengths of the keywords, or the length of the longest keyword) was not found to be helpful in comparing the algorithms.

4.1 Performance versus keyword set size

For each algorithm, the average number of megabytes (of natural language input string) processed per second was graphed against the size of the keyword set. The four graphs (corresponding to AC-FAIL, AC-OPT, CW-BM, and CW-NORM) are superimposed in Figure 1.

As predicted, the AC-OPT algorithm has performance independent of the keyword set size. The AC-FAIL algorithm has slightly worse performance, with a slight decline as the keyword set size increases. The CW-BM and CW-NORM algorithms perform similarly to one another, with the CW-NORM algorithm performing slightly better (in accordance with the theoretical predictions in [WZ92]). The performance of both CW algorithms decreases noticeably with increasing keyword set sizes, eventually being outperformed by the AC-OPT algorithm at keyword set sizes greater than 13.

Figure 2 presents the ratio of the CW-BM performance to the CW-NORM performance — clearly showing CW-NORM outperforming CW-BM. The figure also shows that the performance gap between the two algorithms widens somewhat with increasing keyword set size.

The AC algorithms displayed little to no variance in performance for a given keyword set size. The median performance data (with +1 and −1 standard deviation bars) for AC-FAIL are shown in Figure 3, while those for AC-OPT are shown in Figure 4. In both graphs, the standard

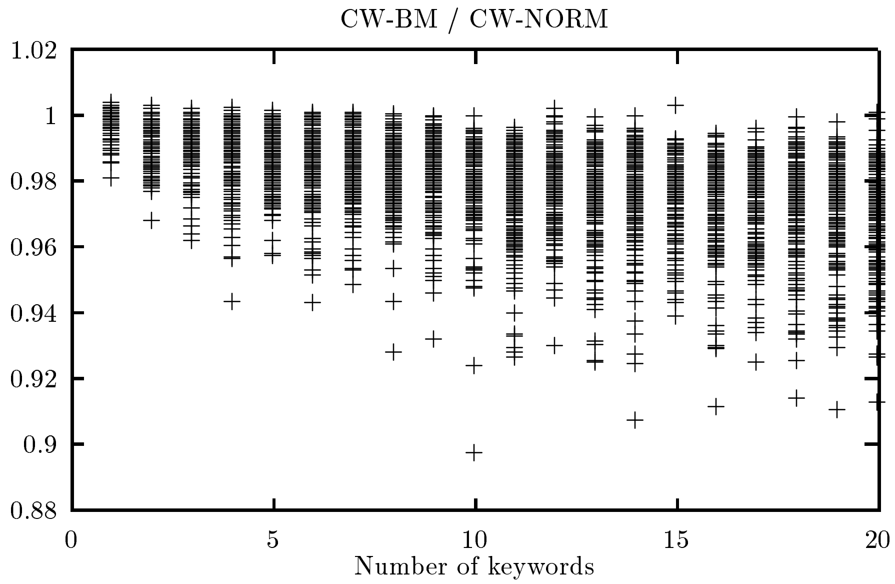

Figure 2: The ratio of CW-BM performance to CW-NORM performance versus keyword set size. Some data-points are greater than 1.00 (although theoretically this should not occur), reflecting timing anomalies due to the limited timer resolution.

deviation bars are very close to the median, indicating that both algorithms display very consistent performance for a given keyword set size.

The CW algorithms displayed a large variance in performance, as is shown in Figures 5 and 6 (for CW-BM and CW-NORM, respectively). These figures show a noticeable narrowing of the standard deviation bars as keyword set size increases. The variance in both algorithms is almost entirely due to the variance in minimum keyword length for a given keyword set size.

For each algorithm, the average number of megabytes of DNA input string processed per second was graphed against keyword set size. The results are superimposed in Figure 7. The performance of the algorithms on the DNA data was similar to their performance on the natural language input data. The performance of the AC-OPT algorithm was independent of the keyword set size, while the performance of the AC-FAIL algorithm declined slightly with increasing keyword set size.

The performance of the CW algorithms, which declined with increasing keyword set size, was consistently better than the AC-OPT algorithm. In some cases, the CW-NORM algorithm displayed a five to ten-fold improvement over the AC-OPT algorithm.

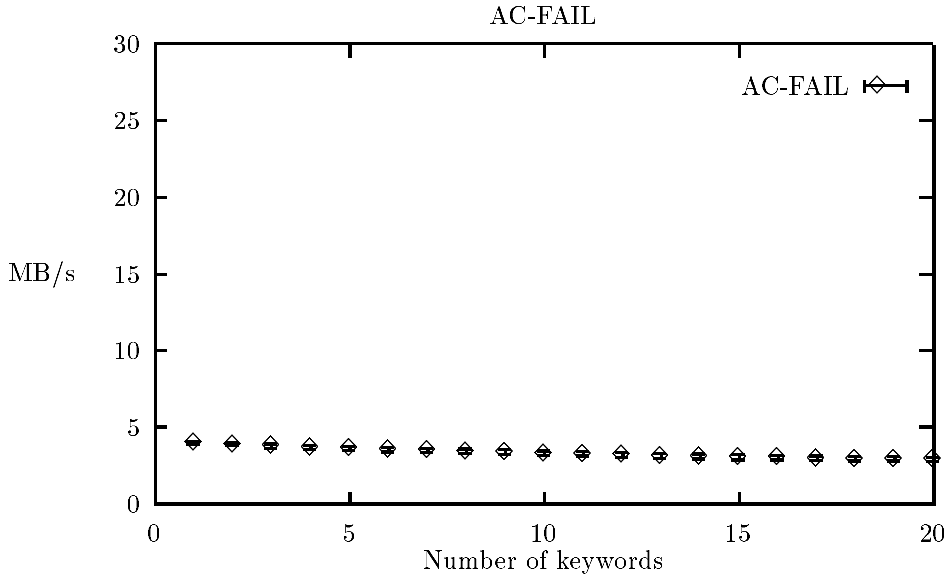

Figure 3: Performance (in megabytes/second) versus keyword set size for the AC-FAIL algorithm. Median performance is shown as a diamond, with +1 and −1 standard deviation bars.

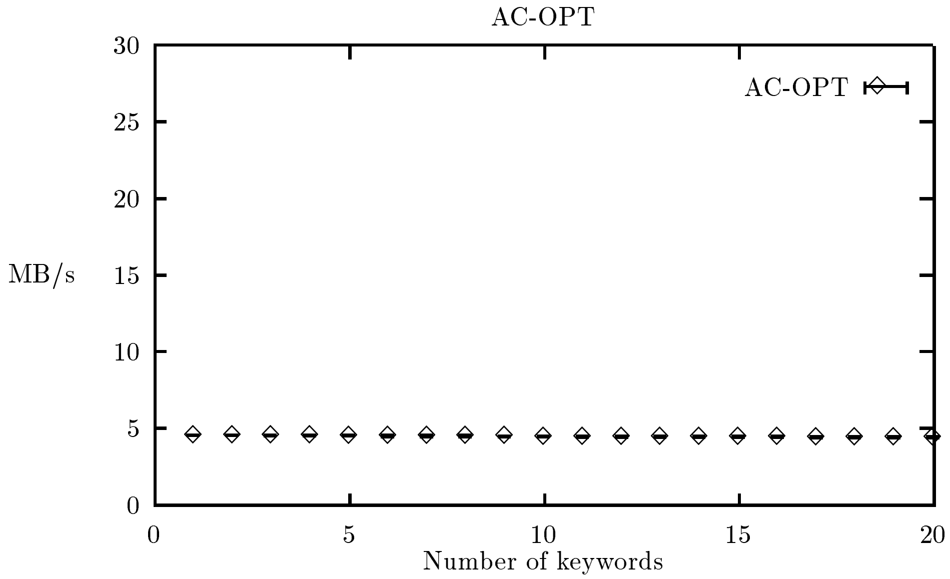

Figure 4: Performance (in megabytes/second) versus keyword set size for the AC-OPT algorithm.

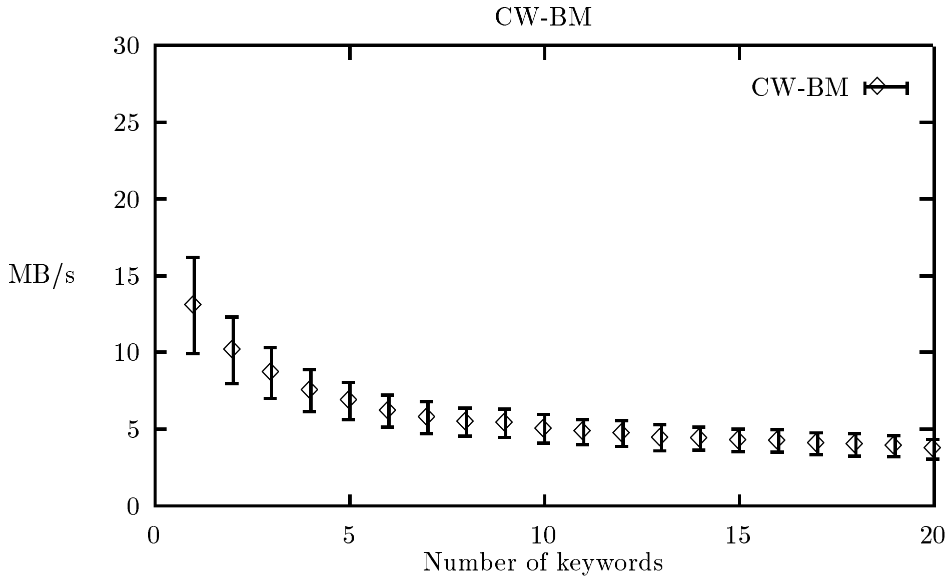

Figure 5: Performance (in megabytes/second) versus keyword set size for the CW-BM algorithm.

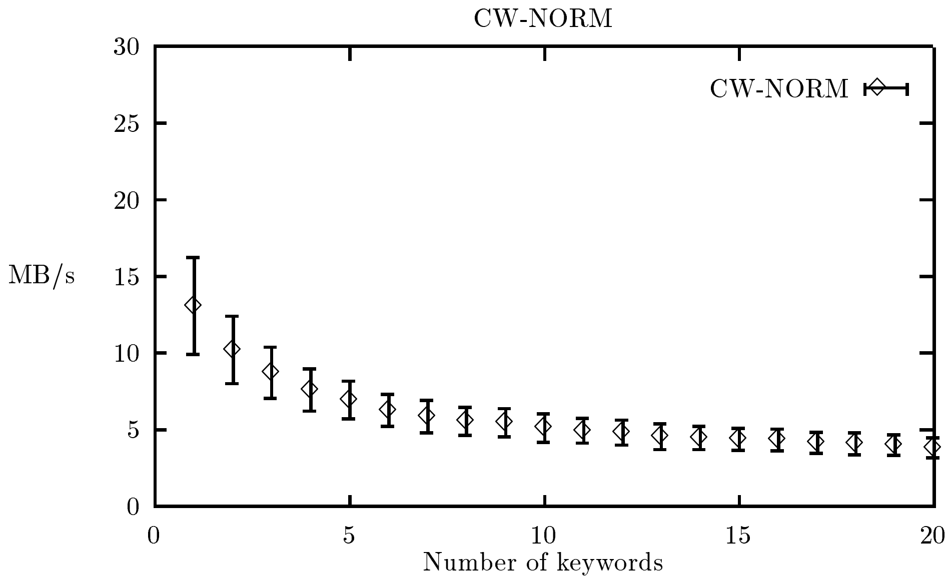

Figure 6: Performance (in megabytes/second) versus keyword set size for the CW-NORM algorithm.

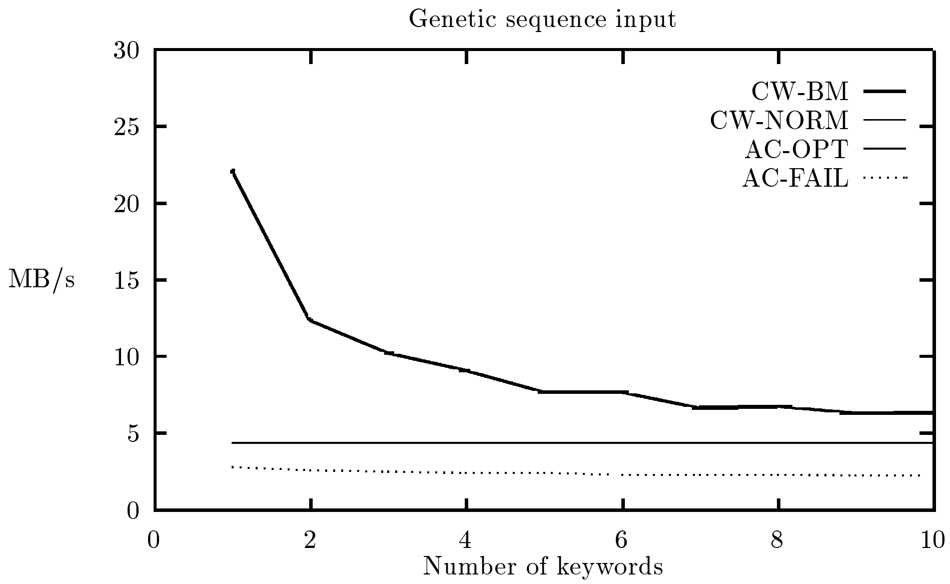

Figure 7: Algorithm performance (in megabytes/second) versus keyword set size, for the DNA test data. The performance of the CW-NORM and CW-BM algorithms are shown superimposed as the descending solid line.

4.2 Performance versus minimum keyword length

For each algorithm, the average number of megabytes processed per second was graphed against the length of the shortest keyword in a set. For the multiple-keyword tests the graphs are superimposed in Figure 8.

Predictably, the AC-OPT algorithm has performance that is independent of the keyword set. The AC-FAIL algorithm has slightly lower performance, improving with longer minimum keywords. The average performance of the CW algorithms improves almost linearly with increasing minimum keyword lengths. The low performance of the CW algorithms for short minimum keyword lengths is explained by the fact that the CW-BM and CW-NORM shift functions are bounded above by the length of the minimum keyword (see [WZ92]). For sets with minimum keywords no less than than four characters, the CW algorithms outperform the AC algorithms.

As predicted, the CW-NORM algorithm outperforms the CW-BM algorithm. The performance ratio of the CW-BM algorithm to the CW-NORM algorithm is shown in Figure 9. The figure indicates that the performance gap is wide with small minimum keyword lengths, and diminishes with increasing minimum keyword lengths. (This effect of the diminishing performance gap is partly due to the distribution of keyword lengths in the test data.)

The AC-FAIL algorithm displayed some variance in performance, as shown in Figure 10. The apparent greater variance for shorter minimum keyword lengths is partially due to the distribution of keyword lengths in the test data. The AC-OPT algorithm showed practically no variance in performance, as shown in Figure 11.

The CW algorithms displayed a large variance in performance for given minimum keyword lengths. The median performance (with $+1$ and -1 standard deviation bars) of the CW-BM algorithm are shown in Figure 12, while those for CW-NORM are shown in Figure 13. The variance increases with increasing shortest keyword length. At a shortest keyword length of 11, the variance decreases abruptly due to the distribution of the shortest keyword lengths of the keyword sets; there are few keyword sets with shortest keyword length greater than 10 characters. The variance in the performance of the CW algorithms is due to the variance in keyword set size (for any given minimum keyword length).

The performance in megabytes of DNA input string processed per second of each algorithm was also graphed against keyword length[2]. The results are superimposed in Figure 14. The performance of the algorithms on the DNA data was similar (though not as dramatic) to their performance on the natural language input data. The performance of the AC-OPT algorithm was independent of the keyword length. Unlike on the natural language input, the AC-FAIL algorithm displayed no noticeable improvement with increasing keyword length; the performance of the AC-FAIL algorithm was little more than half of the performance of the AC-OPT algorithm.

As in the natural language tests, the performance of the CW algorithms improved with increasing keyword length. The rate of performance increase was considerably less than on the natural language input (see Figure 8). On the DNA input, the CW algorithms displayed median performance at least twice that of the AC-OPT algorithm.

[2] Recall that the keywords in a given keyword set were all of the same length.

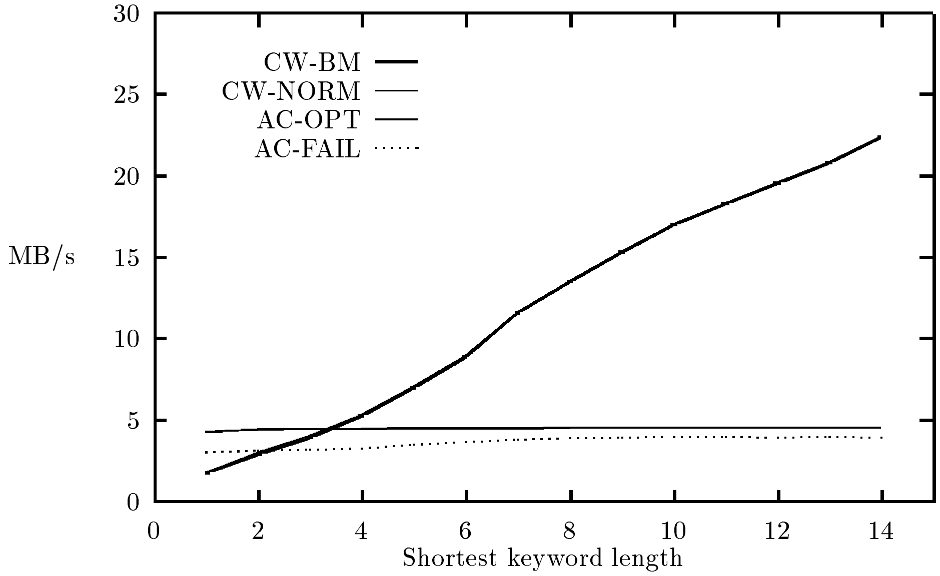

Figure 8: Algorithm performance (in megabytes/second) versus the length of the shortest keyword in a given set. The performance of the CW-BM and CW-NORM algorithms are superimposed (shown as the ascending solid line).

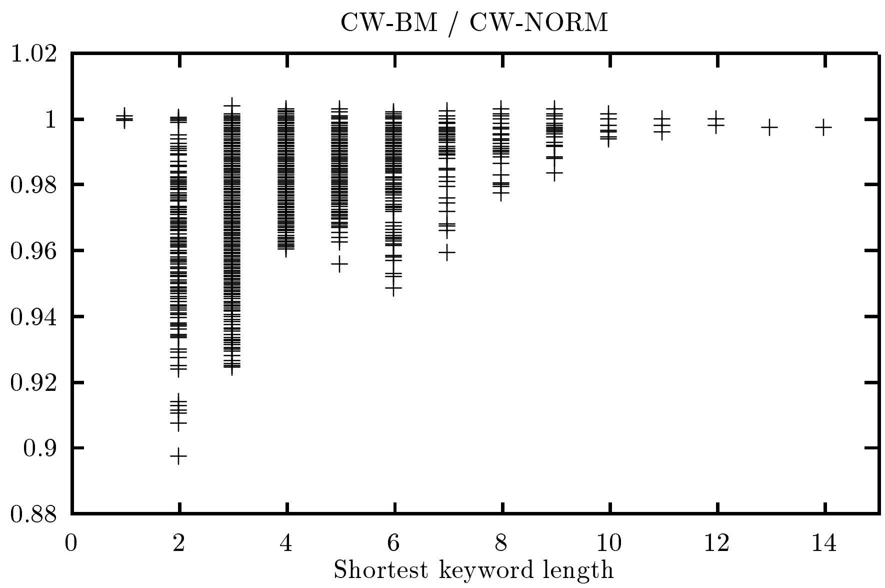

Figure 9: The ratio of CW-BM performance to CW-NORM performance versus the length of the shortest keyword in a set. Some data-points are greater than 1.00, reflecting timing anomalies due to the limited timer resolution.

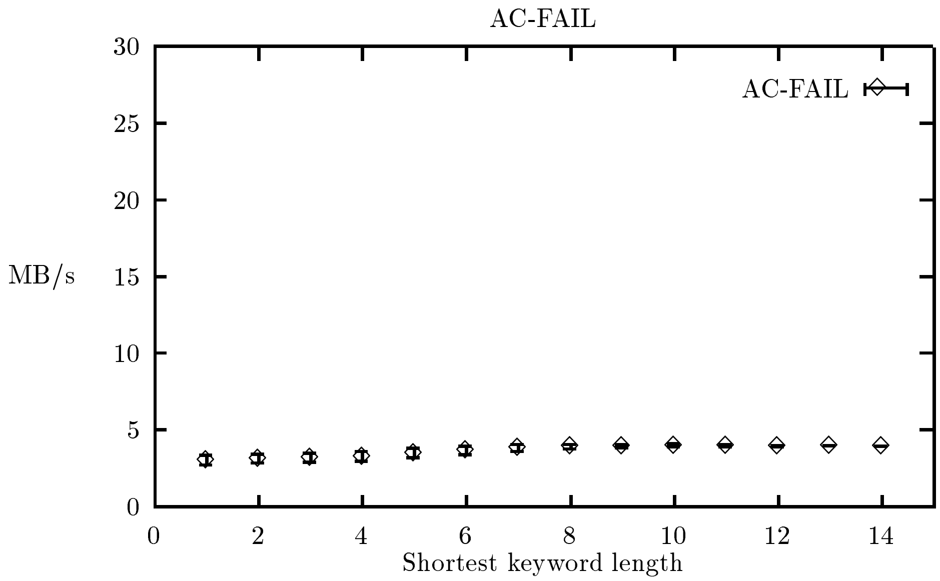

Figure 10: Performance (in megabytes/second) versus the length of the shortest keyword in a set for the AC-FAIL algorithm.

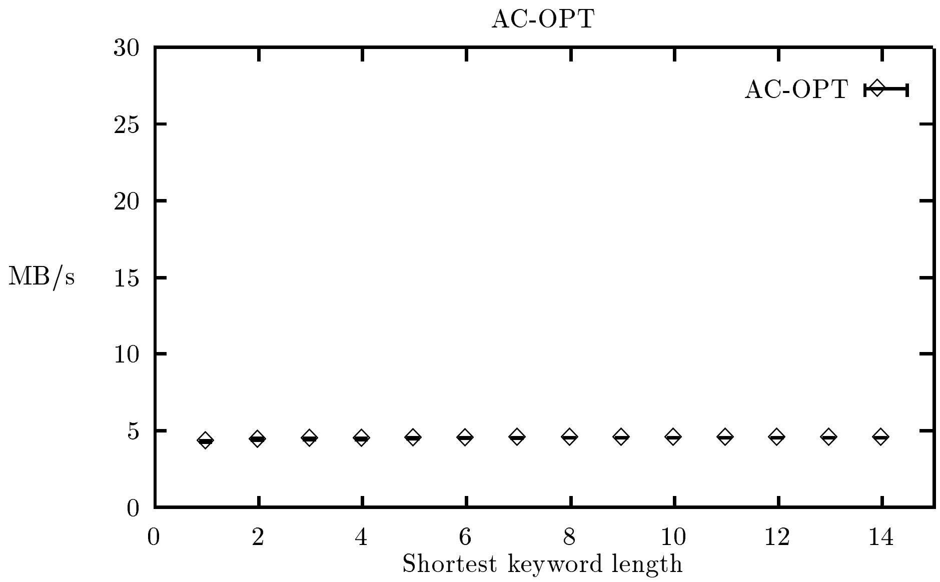

Figure 11: Performance (in megabytes/second) versus the length of the shortest keyword in a set for the AC-OPT algorithm.

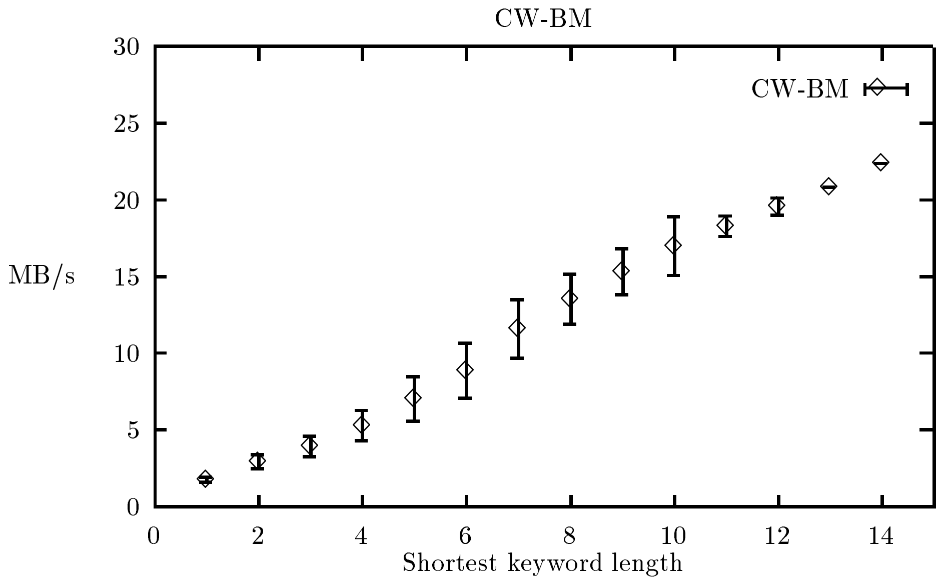

Figure 12: Performance (in megabytes/second) versus the length of the shortest keyword in a set for the CW-BM algorithm.

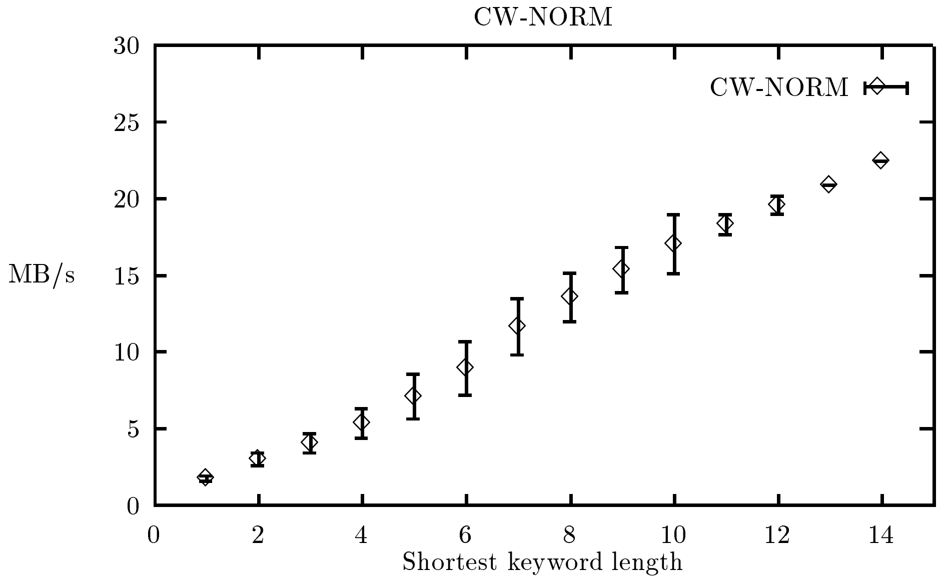

Figure 13: Performance (in megabytes/second) versus the length of the shortest keyword in a set for the CW-NORM algorithm.

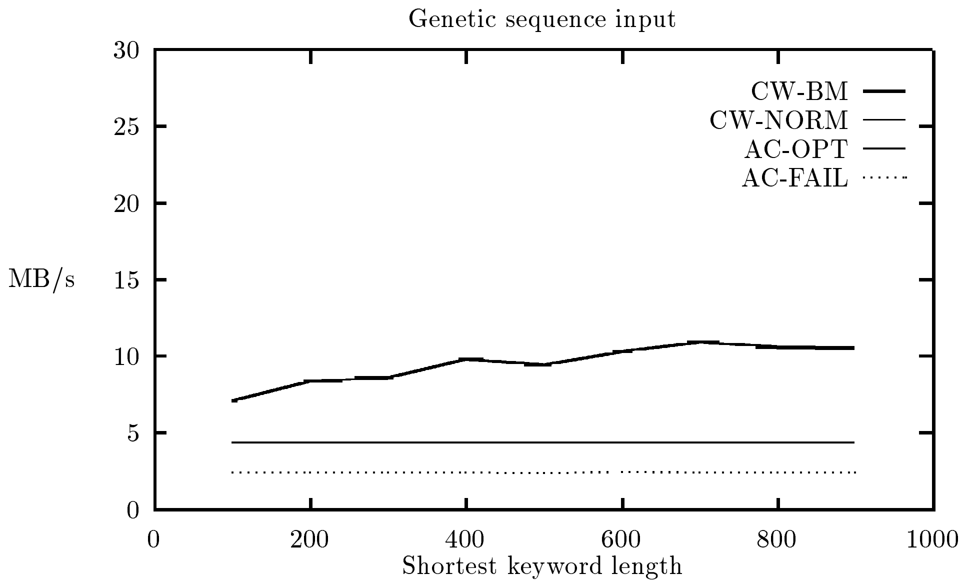

Figure 14: Algorithm performance (in megabytes/second) versus the length of the keywords in a given set, for the DNA test data. The performance of the CW-NORM and CW-BM algorithms are shown superimposed as the ascending solid line.

4.3 Single-keywords

For the single-keyword tests, the average performance (of each algorithm) is graphed against the length of the keyword and superimposed in Figure 15.

The KMP, AC-FAIL, and AC-OPT algorithms displayed performance that was largely independent of the keyword length. The AC-OPT algorithm outperformed the other two, while the KMP algorithm displayed the worst performance. Although the KMP algorithm is similar in structure to the AC-FAIL algorithm, the heavy use of indexing (as opposed to the use of pointers in AC-FAIL) in the KMP algorithm degrades its performance. (The use of indexing makes the KMP algorithm more space efficient than the AC algorithms.) The performance of the CW algorithms improved almost linearly with the length of the keyword, with the CW-NORM algorithm outperforming the CW-BM algorithm.

The variance of the performance of the KMP and the AC-FAIL algorithms was minor, as shown in Figures 16 and 17 (respectively). The AC-OPT algorithm displayed no noticeable variance over the entire range of keyword lengths, as is shown in Figure 18. The CW algorithms showed some variance (increasing with longer keyword lengths) as shown in Figures 19 and 20 respectively.

The performance of the algorithms on the single keyword test data is in agreement with the data collected by Hume and Sunday [HS91].

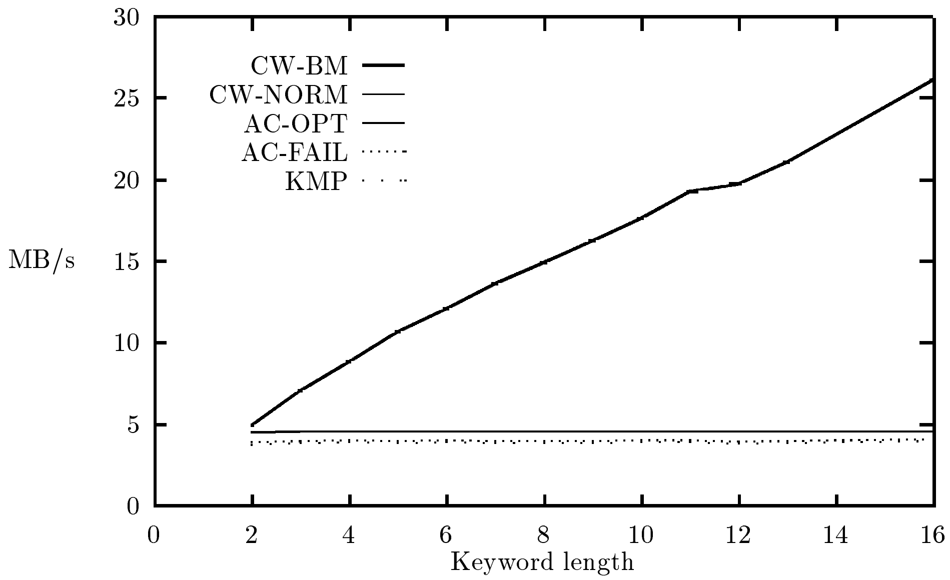

Figure 15: Algorithm performance (in megabytes/second) versus the length of the (single) keyword. The performance of the KMP and AC-FAIL algorithms are shown as the superimposed dotted horizontal line, while those of the CW-BM and CW-NORM algorithms are shown as the superimposed ascending solid line.

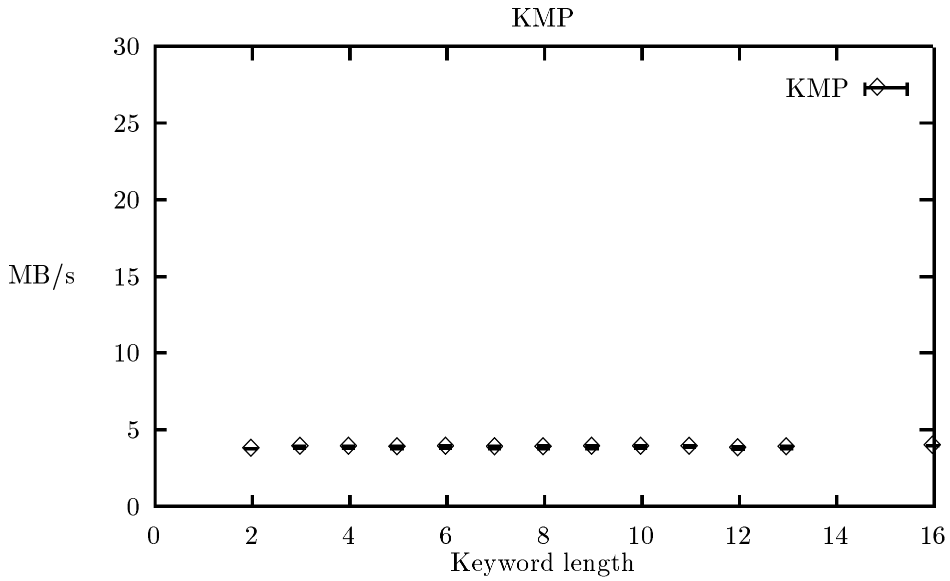

Figure 16: Performance (in megabytes/second) versus the (single) keyword length for the KMP-FAIL algorithm.

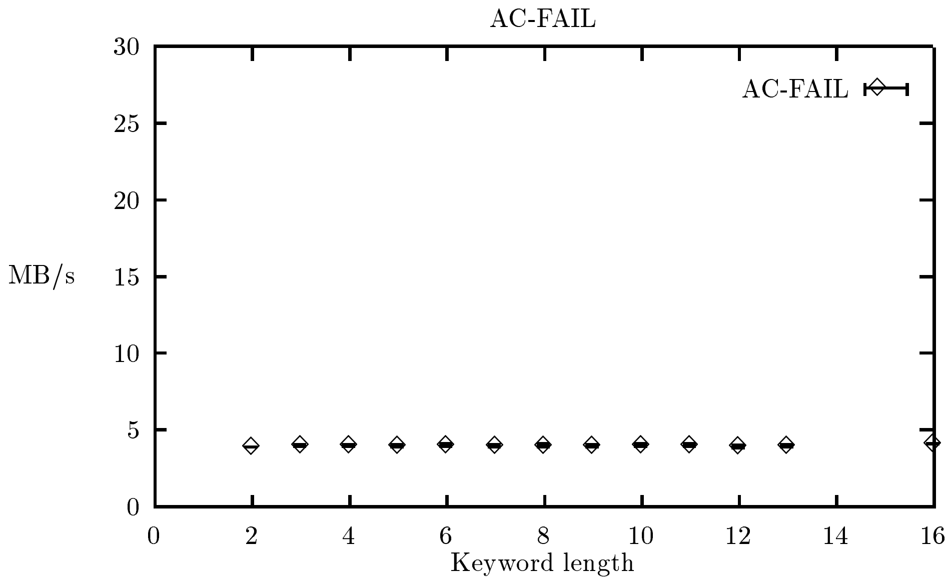

Figure 17: Performance (in megabytes/second) versus the (single) keyword length for the AC-FAIL algorithm.

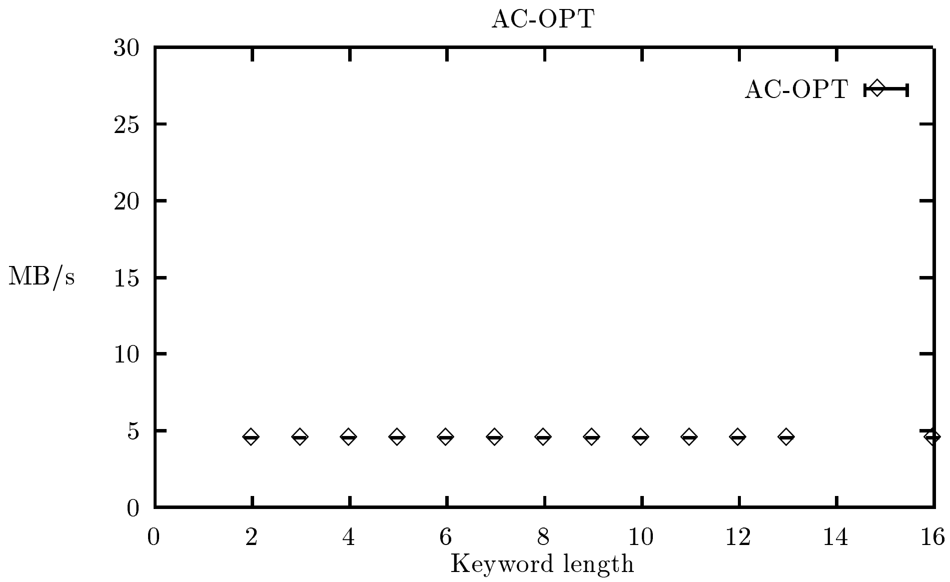

Figure 18: Performance (in megabytes/second) versus the (single) keyword length for the AC-OPT algorithm.

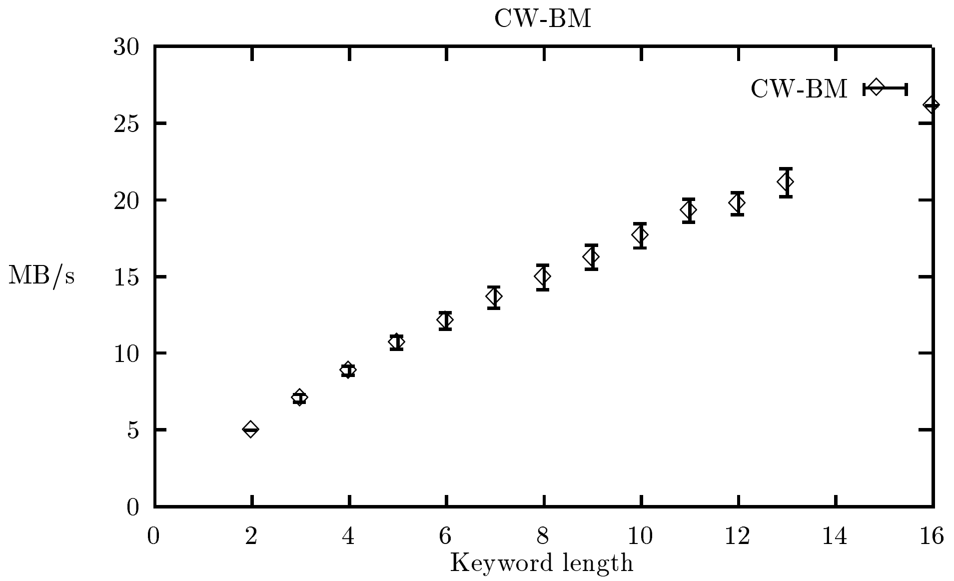

Figure 19: Performance (in megabytes/second) versus the (single) keyword length for the CW-BM algorithm.

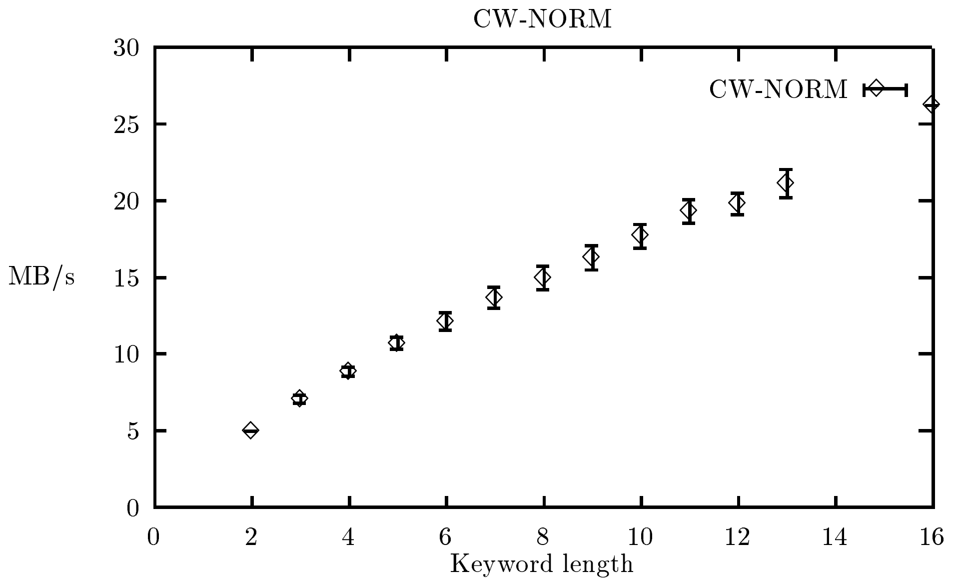

Figure 20: Performance (in megabytes/second) versus the (single) keyword length for the CW-NORM algorithm.

5 Conclusions

The conclusions of this paper fall into two categories: general conclusions regarding the algorithms and testing them, and conclusions relating to the performance of specific algorithms. The general conclusions are:

- The taxonomy of algorithms (appearing in [WZ92]) was crucial to correctly implementing the algorithms in C. In particular, the precomputation algorithm published by Commentz-Walter ([Com79a]) was extremely difficult to understand, while the version appearing in [WZ92] is easy to implement.
- The relative performance of the algorithms did not vary across the testing platforms (the DEC Alpha, HP Snake, and Sun SPARC Station 1+ workstations).
- Testing the algorithms on two vastly differing types of input (English text and DNA sequences) indicates that varying such factors as alphabet size, keyword set size, and smallest keyword length can produce very different rates of performance increase or decrease.
- Comparing algorithm performance to keyword set size and shortest keyword length proved to be more useful (in selecting an algorithm) than comparing performance to other statistics.

The specific performance conclusions are:

- The performance of the AC-OPT algorithm was independent of the keyword sets. The AC-FAIL and KMP algorithm performance increased slightly with increasing shortest keyword length and decreased with increasing keyword set size.
- The performance of the CW algorithms improved approximately linearly with increasing length of the shortest keyword in the keyword set. The rate of increase was much greater with natural language input than with DNA input. The performance of the CW algorithms declined sharply with increasing keyword set size.
- For a given keyword set size and shortest keyword length, the AC and KMP algorithms displayed little or no variance (in performance). The CW algorithms displayed slightly more variance in performance.
- As predicted in [WZ92], the CW-NORM algorithm outperforms the CW-BM algorithm. The cost of precomputation for the two algorithms is approximately the same.
- The AC-OPT algorithm always outperforms the AC-FAIL algorithm; both require similar precomputation, but the AC-FAIL data structures can be made more space efficient.
- On the single-keyword tests:
  - The single-keyword test results were consistent with those presented by Hume and Sunday [HS91].
  - The AC-FAIL algorithm always outperforms the KMP algorithm.
  - In most cases, the CW algorithms outperform the AC algorithms.
- In [Aho90, p. 281], A.V. Aho states that

  > "In practice, with small numbers of keywords, the Boyer-Moore aspects of the Commentz-Walter algorithm can make it faster than the Aho-Corasick algorithm, but with larger numbers of keywords the Aho-Corasick algorithm has a slight edge."

  Although Aho's statement is correct, with the performance data presented in this report, we are able to state more precisely the conditions under which the Commentz-Walter algorithms outperform the Aho-Corasick algorithms.

- On the multiple-keyword natural language tests, the CW algorithms outperformed the AC algorithms when the length of the shortest keyword was long (in general, at least four symbols) and the keyword set size was small (in general, fewer than thirteen keywords). The performance difference between the CW algorithms and the AC algorithms was frequently substantial.

- On the DNA tests, the CW algorithms substantially outperformed the AC algorithms. On these tests the keyword length was at least 100 and the number of keywords in the set was no more than 10. The DNA results show that the CW algorithms can yield much higher performance than the often-used AC-OPT algorithm in areas such as genetic sequence matching.

6 Recommendations

For applications involving small alphabets and long keywords (such as DNA pattern matching), the performance of the CW-NORM algorithm makes it the algorithm of choice. Only when the keyword set size is much larger than ten keywords should the AC-OPT algorithm be considered.

The following procedure can be used to choose a pattern matching algorithm for a natural language pattern matching application:

if performance independent of keyword set is required **then**
  AC-OPT
else
  if multiple-keyword sets are used **then**
    if fewer than thirteen keywords and the shortest keyword length is at least four **then**
      CW-NORM
    else
      choose an AC algorithm
  else (single-keyword sets)
    if space is severely constrained **then**
      KMP
    else
      if the keyword length is at least two **then**
        CW-NORM
      else
        choose an AC algorithm

An AC algorithm can be chosen as follows:

if space efficiency is needed **then**
  AC-FAIL
else
  AC-OPT

A The algorithm toolkit

In this appendix, we describe a toolkit of algorithms for keyword pattern matching. The algorithms are C language implementations of those derived (and proven correct) in the algorithm taxonomy appearing in [WZ92]. The abstract versions contained in the taxonomy are useful for understanding the C versions presented here. Details on obtaining the algorithms are given in Subsection A.5.

The algorithms are the Aho-Corasick failure-function and optimized algorithms (AC-FAIL and AC-OPT respectively), the Knuth-Morris-Pratt algorithm (KMP), the multiple-keyword Boyer-Moore algorithm (CW-BM) and the Commentz-Walter algorithm (CW-NORM). The definition of the C programming language appears in [ISO90].

The algorithms have been implemented primarily for speed, with little concern for memory usage. For example, in the Aho-Corasick failure-function and Commentz-Walter algorithms, a trie datatype is required; each trie has been implemented as large array, which is efficient to access. They could have been implemented using more space efficient techniques (which are less efficient to access), such as those described in [AMS92].

An example of a practical implementation follows. The abstract version of the Aho-Corasick failure-function algorithm (from [WZ92]) is:

$$
\begin{array}{l}
u, r, q := \varepsilon, S, \varepsilon;\ O_e := Output(q) \times \{S\}; \\
\textbf{do}\ r \neq \varepsilon \longrightarrow \\
\qquad \textbf{do}\ \tau_{ef}(q, r{\upharpoonright}1) = \bot_s \longrightarrow q := f_f(q)\ \textbf{od}; \\
\qquad u, r, q := u(r{\upharpoonright}1), r{\downharpoonright}1, \tau_{ef}(q, r{\upharpoonright}1); \\
\qquad O_e := O_e \cup Output(q) \times \{r\} \\
\textbf{od}\ \{R_e\}
\end{array}
$$

(Functions *Output*, τ_{ef}, and f_f are precomputed functions.) Most abstract variables are encoded either as integers or as pointers in the C versions. Variables that undergo such a change of domain (in the C algorithms) are renamed to include the substring *hat*. For example, the variable r in the abstract algorithm is encoded as an integer index and renamed *r_hat* in the C version. The corresponding C version of the algorithm is:

File `acflidx.c`:

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

occ_idx_t *ac_fail_idx( const char S[],
                const kw_t *const ACOutput[],
                const pref_t tau_ef[][V_SIZE],
                occ_idx_t *O_hat_e,
                const pref_t f_f[] ) {

        auto int r_hat;
        auto pref_t q;

        r_hat = 0;
        q = EPSILON;
        Reg_occ(O_hat_e,ACOutput[q],r_hat);
        while( S[r_hat] != 0 ) {
                while( tau_ef[q][S[r_hat]] == BOTTOM ) {
                        q = f_f[q];
                }
                q = tau_ef[q][S[r_hat]];
                r_hat++;
                Reg_occ(O_hat_e,ACOutput[q],r_hat);
        }

        return( O_hat_e );
```

```
}
```

Efforts were made to write readable C code. In several places, constructs could have been replaced with more cryptic (and supposedly more efficient) C idioms. Whether such obfuscation would make the code more efficient was tested: the optimizing C compilers used were able to optimize the readable C code to the same level as the obfuscated C code.

Throughout this appendix, we assume that P is the finite, non-empty set of keyword patterns. The only definitions required are:

$$\begin{array}{rcl} \mathbf{pref}(P) & = & \{x : (\exists\, y : xy \in P)\} \\ \mathbf{suff}(P) & = & \{y : (\exists\, x : xy \in P)\} \end{array}$$

(The set $\mathbf{pref}(P)$ is the set of all words which are prefixes of some word in P, while $\mathbf{suff}(P)$ is the set of all words which are suffixes of some word in P.) Note that: $\epsilon \in \mathbf{pref}(P)$ and $\epsilon \in \mathbf{suff}(P)$.

This appendix is structured as follows: Subsection A.1 presents general data-structures used in the algorithms; Subsection A.2 presents the AC family of algorithms, and their precomputation algorithms; Subsection A.3 gives the KMP algorithm and its precomputation, while Subsection A.4 gives the CW algorithms and their precomputation algorithms; Subsection A.5 gives some information on obtaining and compiling the algorithms.

A.1 General data-structures

The header file `pattm.h` defines a pair of types and some constants. Elements of $\mathbf{pref}(P)$ and $\mathbf{suff}(P)$ are represented by types *pref_t* and *suff_t*, which are defined to be integers. The empty string ϵ (an element of $\mathbf{pref}(P)$ and of $\mathbf{suff}(P)$ since $P \neq \varnothing$) is encoded as *EPSILON*, defined as 0. A special value *BOTTOM* is defined as -1. Its length (as a string) is defined to be -1. The alphabet is defined to be the sequence of characters from 0 up to $V\_SIZE - 1$.

File `pattm.h`:

```
/* (c) Copyright 1993 by Bruce W. Watson */
#ifndef PATTM_H
#define PATTM_H
#include <stdio.h>
#include <limits.h>

/* pref(P) and suff(P) are encoded as integers */
typedef int pref_t;
typedef int suff_t;
#define EPSILON (0)
#define BOTTOM (-1)
#define LEN_OF_BOTTOM (-1)
#define V_SIZE (CHAR_MAX)

#include "kw.h"
#include "occ.h"

#endif
```

The set of keywords is represented using a linked list of structures. The header file `kw.h` declares the structure and defines some macros for operating on the structures. The user of these structures is expected to ensure that no two keywords on the linked list are the same, and so the list implements the set datatype. The user is also expected to ensure that when the empty string (ϵ) is a keyword it appears at the beginning of the list.

Each structure contains the following:

- A pointer p to the string representation of the keyword.

- The length (*length*) of the keyword (length zero is permitted, representing the empty string ϵ).
- The encoding of the keyword P (as a *pref_t* or a *suff_t*) is stored in *end_pref_or_suff*.
- A pointer (*output_next*) to a keyword. Each keyword is maintained on another linked list. Only in the Aho-Corasick algorithm can this list be longer than one element. When a keyword match is found by the Aho-Corasick algorithm, the other keywords on the list are also matches. For more on this, see [WZ92].

File `kw.h`:

```
/* (c) Copyright 1993 by Bruce W. Watson */
#ifndef KW_H
#define KW_H

/* store a keyword as a linked list, with length info, and output function info. */
typedef struct struct_kw_t {
        char *p;
        int length;
        pref_t end_pref_or_suff;
        struct struct_kw_t *next;
        struct struct_kw_t *output_next;
} kw_t;

#define Empty(x) ((x)==NULL)
#define Next(x) ((x)->next)
/* Assume that the epsilon (empty string) keyword is always at the beginning of the linked list */
#define Nonepsilon(x) ( ((x)==NULL) ? (NULL) : ( ((x)->length) ? (x) : ((x)->next) ) )

#endif
```

Keyword matches are registered using the structure declared in header file `occ.h`. There are two versions of each of the four pattern matching algorithms: one using indexing into the input string, and the other using a pointer into the input string. Correspondingly, when a match is found, it is identified using either an integer index, or a pointer. The Knuth-Morris-Pratt algorithm is designed to deal only with a single keyword. As such, it is not necessary to identify the keyword with each match — each match is identified only by an integer index or a pointer (as the case may be). The other algorithms store a structure for each match: an integer index or a pointer into the input string, and a pointer to the keyword that was matched[3]. The macro *Reg_occ* is used by the multiple-keyword algorithms to register a match, if there is one.

File `occ.h`:

```
/* (c) Copyright 1993 by Bruce W. Watson */
#ifndef OCC_H
#define OCC_H

/* pattern match occurrences are stores in an array of some structure. */

typedef struct {
        int where;
        const kw_t *what;
} occ_idx_t;

typedef struct {
        const char *where;
        const kw_t *what;
} occ_ptr_t;
```

[3]For the Aho-Corasick algorithms, this is a pointer to the head of the list (on the linked list *output_next*) of keywords that match.

```
#define Reg_occ(d,rwhat,rwhere) if(rwhat!=NULL){d->where=rwhere;(d++)->what=rwhat;}

typedef int occ_kmp_bm_idx_t;
typedef const char *occ_kmp_bm_ptr_t;

#endif
```

The precomp.h header file contains a constant (representing $+\infty$) and macros computing the **max** and **min** of two numbers.

File precomp.h:

```
/* (c) Copyright 1993 by Bruce W. Watson */
#ifndef PRECOMP_H
#define PRECOMP_H

/* need +infinity for some precomputation */
#define INFINITY (1000000000L)

#define max(x,y) ((x)>(y)?(x):(y))
#define min(x,y) ((x)<(y)?(x):(y))

#endif
```

A.2 The Aho-Corasick algorithms

The AC-FAIL algorithm (functions *ac_fail_idx* and *ac_fail_ptr* in files acflidx.c and acflptr.c, for the indexing and pointer versions respectively) takes five parameters:

- The input string S.
- The array *ACOutput*. (As usual in C, we mean that *ACOutput* is a pointer to the first element of the array.) It maps a *pref_t* (the index type) to the corresponding keyword pointer, used to register a match.
- The two dimensional array *tau_ef*, implementing a trie. It maps a *pref_t* and a character (in the alphabet) to a *pref_t* (possibly the value *BOTTOM*, meaning "undefined"), indicating the next state.
- The pointer *O_hat_e*; this is a pointer to an array of structures used to register matches. The array is used to register keyword matches, and it is assumed to be preallocated of sufficient size to contain all of the matches that may be found in input string S.
- The array *f_f*. It maps a *pref_t* to a *pref_t* (the failure function value).

The return value of the function is a pointer to the next available match registration structure.

File acflidx.c:

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

occ_idx_t *ac_fail_idx( const char S[],
                const kw_t *const ACOutput[],
                const pref_t tau_ef[][V_SIZE],
                occ_idx_t *O_hat_e,
                const pref_t f_f[] ) {
```

```
        auto int r_hat;
        auto pref_t q;

        r_hat = 0;
        q = EPSILON;
        Reg_occ( O_hat_e, ACOutput[q], r_hat );
        while( S[r_hat] != 0 ) {
                while( tau_ef[q][S[r_hat]] == BOTTOM ) {
                        q = f_f[q];
                }
                q = tau_ef[q][S[r_hat]];
                r_hat++;
                Reg_occ( O_hat_e, ACOutput[q], r_hat );
        }

        return( O_hat_e );
}
```

File `acflptr.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

occ_idx_t *ac_fail_ptr( const char *S,
                const kw_t *const ACOutput[],
                const pref_t tau_ef[][V_SIZE],
                occ_ptr_t *O_hat_e,
                const pref_t f_f[] ) {

        auto pref_t q;

        q = EPSILON;
        Reg_occ( O_hat_e, ACOutput[q], S );
        while( *S != 0 ) {
                while( tau_ef[q][*S] == BOTTOM ) {
                        q = f_f[q];
                }
                q = tau_ef[q][*S];
                S++;
                Reg_occ( O_hat_e, ACOutput[q], S );
        }

        return( O_hat_e );
}
```

The AC-OPT algorithm takes four parameters, three of which are as in AC-FAIL (described above). The other parameter, *gamma_f*, is an array. It maps a *pref_t* and a character (from the alphabet) to a *pref_t* (the next state). The return value is as in the failure-function algorithm.

File `acoptidx.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

occ_idx_t *ac_opt_idx( const char S[],
                const kw_t *const ACOutput[],
                const pref_t gamma_f[][V_SIZE],
                occ_idx_t *O_hat_e ) {

        auto int r_hat;
        auto pref_t q;
```

```
        r_hat = 0;
        q = EPSILON;
        Reg_occ( O_hat_e, ACOutput[q], r_hat );
        while( S[r_hat] != 0 ) {
                q = gamma_f[q][S[r_hat]];
                r_hat++;
                Reg_occ( O_hat_e, ACOutput[q], r_hat );
        }

        return( O_hat_e );
}
```

File `acoptptr.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

occ_idx_t *ac_opt_ptr( const char *S,
                const kw_t *const ACOutput[],
                const pref_t gamma_f[][V_SIZE],
                occ_ptr_t *O_hat_e ) {

        auto pref_t q;

        q = EPSILON;
        Reg_occ( O_hat_e, ACOutput[q], S );
        while( *S != 0 ) {
                q = gamma_f[q][*S];
                S++;
                Reg_occ( O_hat_e, ACOutput[q], S );
        }

        return( O_hat_e );
}
```

A.2.1 Aho-Corasick precomputation

File `ptauef.c` contains the precomputation function *tau_ef_precomp*. It takes three parameters:

- The pointer P to the beginning of the keyword set linked list.
- The two dimensional array *tau_ef*. This array is assumed to be preallocated of the appropriate size: the first dimension is of size $|\mathbf{pref}(P)|$, and the second is the size (*V_SIZE*) of the alphabet.
- The array *ACOutput*. The array is assumed to be preallocated of size $|\mathbf{pref}(P)|$.

The trie function (for use with AC-FAIL) is precomputed into the *tau_ef* array, the output function is partially[4] precomputed into the *ACOutput* array. The function returns $|\mathbf{pref}(P)| - 1$.

File `ptauef.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

/*
        Returns |pref(P)| - 1. Assume that tau_ef, ACOutput are |pref(P)|.
        ACOutput is only precomputed for pref's corresponding to EPSILON and keywords.
```

[4] It is only defined on *EPSILON*, and the keywords themselves, not for other elements of $\mathbf{pref}(P)$.

```
*/
int tau_ef_precomp( kw_t *P,
                pref_t tau_ef[][V_SIZE],
                const kw_t *ACOutput[] ) {

        auto pref_t last_pref;
        auto pref_t u;
        auto char a;
        auto const char *p;
        auto int v_hat;

        last_pref = EPSILON;
        for( a = 0; a < V_SIZE; a++ ) {
                tau_ef[EPSILON][a] = BOTTOM;
        }
        ACOutput[EPSILON] = NULL;
        while( !Empty(P) ) {
                p = P->p;
                u = EPSILON;
                v_hat = 0;

                while( v_hat < P->length ) {
                        if( tau_ef[u][p[v_hat]] == BOTTOM ) {
                                tau_ef[u][p[v_hat]] = ++last_pref;
                                for( a = 0; a < V_SIZE; a++ ) {
                                        tau_ef[last_pref][a] = BOTTOM;
                                }
                                ACOutput[last_pref] = NULL;
                        }
                        u = tau_ef[u][p[v_hat]];
                        v_hat++;
                }
                P->end_pref_or_suff = u;
                ACOutput[u] = P;
                P->output_next = NULL;
                P = Next(P);
        }
        for( a = 0; a < V_SIZE; a++ ) {
                if( tau_ef[EPSILON][a] == BOTTOM ) {
                        tau_ef[EPSILON][a] = EPSILON;
                }
        }
        return( last_pref );
}
```

The file `pgamma-f.c` contains the precomputation function *gamma_f_precomp*. It takes five parameters:

- The *gamma_f* array. This array is assumed to be preallocated of the appropriate size, with the first dimension of size $|\mathbf{pref}(P)|$, and the second dimension is of size *V_SIZE*.
- The *f_f* array. This array is assumed to be preallocated of size $|\mathbf{pref}(P)|$.
- The *ACOutput* and *tau_ef* arrays. These arrays are assumed to be precomputed (partially, in the case of *ACOutput*) by function *tau_ef_precomp*.
- The *scratch* array. This array is assumed to be preallocated of size $|\mathbf{pref}(P)|$. It is used as an auxiliary variable in performing a breadth-first traversal of the trie *tau_ef*.

The function computes the transition function into the array *gamma_f*, and the failure-function into array *f_f*. The function has no return value.

File `pgamma-f.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

/*
	Expects to receive tau_ef already precomputed.
	Assume ACOutput precomputed for EPSILON and keywords.
*/
void gamma_f_precomp( pref_t gamma_f[][V_SIZE],
		pref_t f_f[],
		kw_t *ACOutput[],
		const pref_t tau_ef[][V_SIZE],
		pref_t scratch[] ) {

	auto pref_t u;
	auto pref_t ua;
	auto char a;
	auto int ctr1;
	auto int ctr2;

	u = EPSILON;
	for( a = 0; a < V_SIZE; a++ ) {
		gamma_f[EPSILON][a] = tau_ef[EPSILON][a];
		f_f[gamma_f[EPSILON][a]] = EPSILON;
		if( gamma_f[EPSILON][a] != EPSILON ) {
			/* a in pref(P) */
			/* set up for BFS */
			scratch[ctr1++] = gamma_f[EPSILON][a];
		}
	}
	scratch[ctr1] = BOTTOM;

	for( ctr2 = 0; scratch[ctr2] != BOTTOM; ctr2++ ) {
		/* update for next BFS */
		u = scratch[ctr2];
		for( a = 0; a < V_SIZE; a++ ) {
			if( tau_ef[u][a] != BOTTOM ) {
				scratch[ctr1++] = tau_ef[u][a];
			}
		}
		scratch[ctr1] = BOTTOM;

		if( ACOutput[u] == NULL ) {
			ACOutput[u] = ACOutput[f_f[u]];
		} else {
			/* u is a keyword */
			u->output_next = ACOutput[f_f[u]];
		}
		for( a = 0; a < V_SIZE; a++ ) {
			ua = tau_ef[u][a];

			if( ua != BOTTOM ) {
				gamma_f[u][a] = ua;
				f_f[ua] = gamma_f[f_f[u]][a];
			} else {
				gamma_f[u][a] = gamma_f[f_f[u]][a];
			}
		}
	}
	return;
}
```

Since $|\mathbf{pref}(P)| \leq \Sigma_{p \in P}|p|$, we can use $\Sigma_{p \in P}|p|$ as an estimate for $|\mathbf{pref}(P)|$ when allocating memory.

A.3 The Knuth-Morris-Pratt algorithm

The KMP algorithm can deal with only a single-keyword as the pattern. Like the other pattern matching algorithms, it comes in two versions: using integer indexing, and using pointers. It takes five parameters:

- The input string S.
- The pointer *O_hat_kmp*. It points to an array, used to register matches. The array pointed to is assumed to be preallocated large enough to contain all of the matches.
- The array *f_hat_ef*. The array maps an integer in the range $[0, |p|)$ (where p is the keyword) to an integer (representing the next state).
- The keyword string p. Note that the keyword is not a *kw_t* structure.
- The keyword length *p_length*.

The function returns a pointer to the next available match structure.

File `kmpidx.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

occ_kmp_bm_idx_t *kmp_idx( const char S[],
                    occ_kmp_bm_idx_t *O_hat_kmp,
                    const int f_hat_ef[],
                    const char p[],
                    const int p_length ) {

        auto int i;
        auto int j;

        i = 0;
        j = 0;
        if( i == p_length ) {
                *O_hat_kmp = j;
                O_hat_kmp++;
        }
        while( S[j] != 0 ) {
                while( 0 <= i && S[j] != p[i] ) {
                        i = f_hat_ef[i];
                }
                i++;
                j++;
                if( i == p_length ) {
                        *O_hat_kmp = j;
                        O_hat_kmp++;
                }
        }

        return( O_hat_kmp );
}
```

File `kmpptr.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"
```

```
occ_kmp_bm_ptr_t *kmp_ptr( const char *S,
                occ_kmp_bm_ptr_t *O_hat_kmp,
                const int f_hat_ef[],
                const char p[],
                const int p_length ) {

        auto const char *j;
        auto int i;

        j = S;
        i = 0;
        if( i == p_length ) {
                *O_hat_kmp = j;
                O_hat_kmp++;
        }
        while( *j != 0 ) {
                while( 0 <= i && *j != p[i] ) {
                        i = f_hat_ef[i];
                }
                i++;
                j++;
                if( i == p_length ) {
                        *O_hat_kmp = j;
                        O_hat_kmp++;
                }
        }

        return( O_hat_kmp );
}
```

A.3.1 Knuth-Morris-Pratt precomputation

The precomputation for the KMP algorithm is performed by a function *f_hat_ef_precomp* (appearing in file `pfht-ef.c`), taking three parameters:

- The keyword string *p*.
- The keyword length *p_len*.
- The array *f_hat_ef*. The array is assumed to be preallocated of at least the length of the keyword.

The KMP failure-function is computed into the array *f_hat_ef*. The function has no return value.

File `pfht-ef.c`:

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

void f_hat_ef_precomp( const char p[],
                const int p_len,
                int f_hat_ef[] ) {

        f_hat_ef[0] = LEN_OF_BOTTOM;
        if( p[0] != 0 ) {
                auto int v_hat = 1;

                f_hat_ef[v_hat] = 0;
                while( p[v_hat] != 0 ) {
                        auto int u_hat_prime = f_hat_ef[v_hat];

                        while( p[u_hat_prime] != p[v_hat] && u_hat_prime != 0 ) {
                                u_hat_prime = f_hat_ef[u_hat_prime];
```

```
                        }
                        if( p[u_hat_prime] == p[v_hat++] ) {
                                f_hat_ef[v_hat] = u_hat_prime + 1;
                        } else {
                                f_hat_ef[v_hat] = 0;
                        }
                }
        }
}
```

A.4 The Commentz-Walter algorithms

The CW algorithm skeleton appears in two versions: an integer indexing version, and a pointer version. The CW algorithm skeleton is used with two shift functions, the multiple-keyword CW-BM shift funtion, and the CW normal shift function (for a description of the differences, see [WZ92]). The CW algorithms are unable to deal with the case that ϵ (the empty word) is a keyword. The function takes six parameters:

- The input string *S_prime*. The algorithm requires that *S_prime*$[-1]$ is accessible, and contains the null character (0). It also requires that m null characters are appear at the end of *S_prime*, where m is the length of the shortest keyword in P.
- The array *CWOutput*. It maps a *suff_t* to the corresponding keyword pointer used to register matches.
- The two dimensional array *tau_r*, representing a trie. It maps a *suff_t* and a character (in the alphabet) to a *suff_t* (or *BOTTOM*, meaning "undefined").
- The pointer *O_hat_e* to an array. The array is used to register matches, and it is assumed to be preallocated of sufficient size to contain all of the matches.
- The two dimensional array *CWShift*. It maps a *suff_t* and a character (in the alphabet) to a shift distance. Two such shift arrays are the CW-BM shift function and the CW-NORM shift function.
- The integer m — the length of the shortest keyword in P.

The function returns a pointer to the next available match registration structure.

File `cwidx.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

occ_idx_t *cw_idx( const char S_prime[],
                const kw_t *const CWOutput[],
                const suff_t tau_r[][V_SIZE],
                occ_idx_t *O_hat_e,
                const int CWShift[][V_SIZE],
                const int m ) {

        auto int u_hat;
        auto int l_hat;
        auto suff_t v;

        u_hat = m - 1;
        while( S_prime[u_hat] != 0 ) {
                l_hat = u_hat;
                v = EPSILON;

                while( 0 <= l_hat && tau_r[v][S_prime[l_hat]] != BOTTOM ) {
```

```
                        v = tau_r[v][S_prime[l_hat]];
                        l_hat--;
                        Reg_occ( O_hat_e, CWOutput[v], u_hat+1);
                }
                u_hat += CWShift[v][S_prime[l_hat]];
        }

        return( O_hat_e );
}
```

File `cwptr.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

occ_idx_t *cw_ptr( const char S_prime[],
                const kw_t *const CWOutput[],
                const suff_t tau_r[][V_SIZE],
                occ_idx_t *O_hat_e,
                const int CWShift[][V_SIZE],
                const int m ) {

        auto const char *u;
        auto const char *l;
        auto suff_t v;

        u = S_prime + m - 1;
        while( *u != 0 ) {
                l = u;
                v = EPSILON;

                while( S_prime <= l && tau_r[v][*l] != BOTTOM ) {
                        v = tau_r[v][*l];
                        l--;
                        Reg_occ( O_hat_e, CWOutput[v], u+1);
                }
                u += CWShift[v][*l];
        }

        return( O_hat_e );
}
```

A.4.1 Common precomputation functions

All of the CW precomputation functions assume that ϵ (the empty word) is not a keyword. The function *tau_r_precomp* (in file `ptaur.c`) is the CW precomputation function usually called first. It takes four parameters:

- The pointer P to the beginning of the keyword set linked list.
- The two dimensional array *tau_r*. The first dimension is assumed to be preallocated of size $|\mathbf{suff}(P)|$. The second dimension is assumed to be preallocated of size *V_SIZE*.
- The array *CWOutput*. It is assumed to be preallocated of size $|\mathbf{suff}(P)|$.
- The array *suff_lengths*. It is assumed to be preallocated of size $|\mathbf{suff}(P)|$.

The function returns the last *suff_t* that was used to encode an element of $\mathbf{suff}(P)$.

File `ptaur.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"

/*
	Returns |suff(P)| - 1.
	Assume suff_lengths is scratch memory of size |suff(P)|.
*/
suff_t tau_r_precomp( kw_t *P,
		suff_t tau_r[][V_SIZE],
		const kw_t *CWOutput[],
		int suff_lengths[] ) {

	auto suff_t last_suff;
	auto suff_t u;
	auto char a;
	auto const char *p;
	auto int v_hat;

	last_suff = EPSILON;
	for( a = 0; a < V_SIZE; a++ ) {
		tau_r[EPSILON][a] = BOTTOM;
	}
	suff_lengths[EPSILON] = 0;
	while( !Empty(P) ) {
		p = P->p;
		u = EPSILON;
		v_hat = P->length - 1;

		while( 0 <= v_hat ) {
			if( tau_r[u][p[v_hat]] == BOTTOM ) {
				tau_r[u][p[v_hat]] = ++last_suff;
				suff_lengths[last_suff] = suff_lengths[u] + 1;
				for( a = 0; a < V_SIZE; a++ ) {
					tau_r[last_suff][a] = BOTTOM;
				}
				CWOutput[last_suff] = NULL;
			}
			u = tau_r[u][p[v_hat]];
			v_hat--;
		}
		P->end_pref_or_suff = u;
		CWOutput[u] = P; P->output_next = NULL;
		P = Next(P);
	}
	return( last_suff );
}
```

The function *cw_m_P_precomp* (in file `pcw-m-p.c`) takes one parameter:

- The pointer P to the beginning of the keyword set linked list.

The function returns an integer: the length of the shortest keyword in P.

File `pcw-m-p.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"
#include "precomp.h"

int cw_m_P_precomp( const kw_t *P ) {

	auto int m_P;
```

```
	m_P = INFINITY;
	while( !Empty(P) ) {
		m_P = min(m_P,P->length);
		P = Next(P);
	}
	return( m_P );
}
```

The function *cw_d3_bar_precomp* (in file `pcwd3bar.c`) takes two parameters:

- The pointer P to the beginning of the keyword set linked list.
- The array *d3_bar*. It is assumed to be preallocated of size *V_SIZE*. The function computes a raw shift function into this array.

The function has no return value.

File `pcwd3bar.c`**:**

```
/* (c) Copyright 1993 by Bruce W. Watson */
#include "pattm.h"
#include "precomp.h"

/*
	Expects d3_bar to be allocated.
*/
void cw_d3_bar_precomp( const kw_t *P,
		int d3_bar[V_SIZE] ) {

	auto int i;
	auto char a;

	for( a = 0; a < V_SIZE; a++ ) {
		d3_bar[a] = INFINITY;
	}
	while( !Empty(P) ) {
		/* this correctly starts at 1 */
		for( i = 1; i < P->length; i++ ) {
			if( i < d3_bar[P->p[P->length - i - 1]] ) {
				d3_bar[P->p[P->length - i - 1]] = i;
			}
		}
		P = Next(P);
	}
	return;
}
```

The function *cw_d1_d2_precomp* (in file `pcwd1-d2.c`) is the main precomputation function for the CW-BM and the CW algorithms. The function takes ten parameters:

- Pointer P to the beginning of the keyword set linked list.
- Two dimensional array *gr*. It is assumed to be the *tau_r* array precomputed by function *tau_r_precomp*. The function *cw_d1_d2_precomp* destroys the contents of the array. Since the contents (as computed by *tau_r_precomp*) are required by the CW skeletons, a copy of array *tau_r* should be made, with the copy being passed to *cw_d1_d2_precomp*.
- Array *fr*, used as a scratch array. The array is assumed to be preallocated of size $|\mathbf{suff}(P)|$.
- Array *CWOutput*, assumed to be precomputed by function *tau_r_precomp*.

- Arrays *d1* and *d2*. The function computes raw shift functions into these arrays (which are then further processed by other functions). They are both assumed to be preallocated of size $|\mathbf{suff}(P)|$.
- The *suff_t last_suff*. This is assumed to be the return value of function *tau_r_precomp* — the last *suff_t* value used to encode an element of $\mathbf{suff}(P)$.
- Arrays *scratch* and *right_drop*, both used as scratch arrays (for a breadth-first search). Both are assumed to be preallocated of size $|\mathbf{suff}(P)|$.
- Array *suff_lengths*. The array maps a *suff_t* to the length of the elemement of $\mathbf{suff}(P)$ it encodes. It is assumed to be precomputed by function *tau_r_precomp*.

The function has no return value.

File `pcwd1-d2.c`:

```
/* Copyright (c) 1993 by Bruce W. Watson */
#include "pattm.h"
#include "precomp.h"

/*
	Assume that gr, fr are allocated of size |suff(P)|
*/
void cw_d1_d2_precomp( const kw_t *P,
		suff_t gr[][V_SIZE],
		suff_t fr[],
		const kw_t *const CWOutput[],
		int d1[],
		int d2[],
		const suff_t last_suff,
		suff_t scratch[],
		suff_t right_drop[],
		const int suff_lengths[] ) {

	/* scratch[], right_drop[] must be of length |suff(P)| */
	auto suff_t u;
	auto suff_t au;
	auto suff_t v;
	auto char a;
	auto int ctr1
	auto int ctr2;

	ctr1 = 0;
	for( u = EPSILON; u <= last_suff; u++ ) {
		d1[u] = d2[u] = INFINITY;
	}
	for( a = 0; a < V_SIZE; a++ ) {
		if( gr[EPSILON][a] != BOTTOM ) {
			fr[gr[EPSILON][a]] = EPSILON;
			d1[EPSILON] = 1;
			if( CWOutput[gr[EPSILON][a]] != NULL ) {
				d2[EPSILON] = 1;
			}
			/* Now set up for BFS and right drop 1 of a suff */
			scratch[ctr1++] = gr[EPSILON][a];
			right_drop[gr[EPSILON][a]] = EPSILON;
		} else {
			gr[EPSILON][a] = EPSILON;
		}
	}

	scratch[ctr1] = BOTTOM;
	for( ctr2 = 0; scratch[ctr2] != BOTTOM; ctr2++ ) {
		/* update the next suff's for the BFS */
```

```
        for( a = 0; a < V_SIZE; a++ ) {
            if( gr[scratch[ctr2]][a] != BOTTOM ) {
                scratch[ctr1++] = gr[scratch[ctr2]][a];
                right_drop[gr[scratch[ctr2]][a]] = scratch[ctr2];
            }
        }
        scratch[ctr1] = BOTTOM;

        /* now main part of loop */
        u = scratch[ctr2];
        for( a = 0; a < V_SIZE; a++ ) {
            au = gr[u][a];

            if( au != BOTTOM ) {
                fr[au] = gr[fr[u]][a];
                d1[fr[au]] =
                    min(d1[fr[au]],(suff_lengths[au]-suff_lengths[fr[au]]));
                if( CWOutput[au] != NULL ) {
                    /* au in P */
                    v = fr[au];

                    while( v != EPSILON ) {
                        d2[v] = min(d2[v],(suff_lengths[au]-suff_lengths[v]));
                        v = fr[v];
                    }
                    d2[EPSILON] = min(d2[EPSILON],suff_lengths[au]);
                }
            } else {
                gr[u][a] = gr[fr[u]][a];
            }
        }
    }
    for( ctr2 = 0; scratch[ctr2] != BOTTOM; ctr2++ ) {
        d2[scratch[ctr2]] = min(d2[scratch[ctr2]],d2[right_drop[scratch[ctr2]]]);
    }
    return;
}
```

A.4.2 Boyer-Moore precomputation

The multiple-keyword CW-BM shift function is computed by function *cw_bm_precomp*, appearing in file `pcw-kbm.c`. The function takes six parameters:

- The two dimensional array *k_bm*. The function fills this array with the CW-BM shift function. The first dimension is assumed to be preallocated of size $|\mathbf{pref}(P)|$. The second dimension is assumed to be preallocated of size *V_SIZE*.
- The arrays *d1* and *d2*. Both arrays are assumed to contain raw shift functions (assumed to have been precomputed by function *cw_d1_d2_precomp*). Both arrays map a *suff_t* to an integer.
- The array *Char*. The array is assumed to contain a raw shift function precomputed by function *cw_char_precomp*.
- The *suff_t last_suff*. This is assumed to be the value returned by function *tau_r_precomp*. It is the last value (of *suff_t*) used to encode an element of $\mathbf{suff}(P)$.
- The array *suff_lengths*. The array maps a *suff_t* to the length of the element of $\mathbf{suff}(P)$ encoded by the *suff_t*. It is assumed to be precomputed by function *tau_r_precomp*.

The function does not have a return value.

File `pcw-kbm.c`**:**

```
/* Copyright (c) 1993 by Bruce W. Watson */

#include "pattm.h"
#include "precomp.h"

/*
        Assume k_bm is allocated of size |suff(P)|.
*/
void cw_bm_precomp( int k_bm[][V_SIZE],
                const int d1[],
                const int d2[],
                const int Char[V_SIZE],
                const int suff_lengths[],
                const suff_t last_suff ) {

        auto suff_t u;
        auto char a;

        for( u = EPSILON; u <= last_suff; u++ ) {
                for( a = 0; a < V_SIZE; a++ ) {
                        k_bm[u][a] = max((Char[a]-suff_lengths[u]),(min(d1[u],d2[u])));
                }
        }
        return;
}
```

Function *cw_char_precomp* (in function `pcwchar.c`) takes three parameters:

- The array *d3_bar*. It contains a raw shift function, assumed to be precomputed by function *cw_d3_bar_precomp*.
- The array *Char*, into which the function computes an integer shift. The array is assumed to be preallocated of size *V_SIZE*.
- The integer *m_P* — the length of the shortest keyword in P. It is usually precomputed by function *cw_m_P_precomp*.

It does not have a return value.

File `pcwchar.c`**:**

```
/* Copyright (c) 1993 by Bruce W. Watson */
#include "pattm.h"
#include "precomp.h"

void cw_char_precomp( const int d3_bar[V_SIZE],
                int Char[V_SIZE],
                const int m_P ) {

        auto char a;

        for( a = 0; a < V_SIZE; a++ ) {
                Char[a] = min(d3_bar[a],m_P);
        }
        return;
}
```

A.4.3 Commentz-Walter precomputation

The CW normal shift function is computed by function *cw_bm_precomp* (in file `pcw-kbm.c`). The function takes six parameters:

- The two dimensional array *k_norm*. The function fills this array with the CW normal shift function. The first dimension is assumed to be preallocated of size $|\mathbf{pref}(P)|$. The second dimension is assumed to be preallocated of size *V_SIZE*.
- The arrays *d1* and *d2*. Both arrays contain raw shift functions, assumed to have been precomputed by function *cw_d1_d2_precomp*.
- The array *d3_bar*. The array is assumed to contain a raw shift function precomputed by function *cw_d3_bar_precomp*.
- The *suff_t last_suff*. This is assumed to be the value returned by function *tau_r_precomp*.
- The array *suff_lengths*. It is assumed to be precomputed by function *tau_r_precomp*.

It does not have a return value.

File `pcw-knor.c`:

```
/* Copyright (c) 1993 by Bruce W. Watson */
#include "pattm.h"
#include "precomp.h"

#define d3(z,a) ((d3_bar[(a)] == INFINITY) ? (INFINITY) : (d3_bar[(a)] - (z)))

/*
        Assume k_norm is allocated of size |suff(P)|.
*/
void cw_norm_precomp( int k_norm[][V_SIZE],
                const int d1[],
                const int d2[],
                const int d3_bar[V_SIZE],
                const suff_t last_suff,
                const int suff_lengths[] ) {

        auto char a;
        auto suff_t u;

        for( u = EPSILON; u <= last_suff; u++ ) {
                for( a = 0; a < V_SIZE; a++ ) {
                        k_norm[u][a] =
                                min((max(d3(suff_lengths[u],a),d1[u])),d2[u]);
                }
        }
        return;
}

#undef d3
```

A.5 Obtaining and compiling the algorithms

The algorithms are available for anonymous ftp as tar'd compressed file pattmkit.tar.Z from site ftp.win.tue.nl (also known as 131.155.70.100) in directory pub/techreports/pi/pattm/bench. The taxonomy paper [WZ92] is also available (in the same directory) as file pattm.ps.Z.

The following table gives the source files required for compiling each of the pattern matching algorithms:

Algorithm	*Algorithm code*	*Precomputation code*
AC-FAIL	`acflidx.c` or `acflptr.c`	`ptauef.c` `pgamma-f.c`
AC-OPT	`acoptidx.c` or `acoptptr.c`	`ptauef.c` `pgamma-f.c`
KMP	`kmpidx.c` or `kmpptr.c`	`pfht-ef.c`

CW-BM	`cwidx.c` or `cwptr.c`	`ptaur.c`
		`pcw-m-p.c`
		`pcwd3bar.c`
		`pcwd1d2.c`
		`pcwchar.c`
		`pcw-kbm.c`

CW-NORM	`cwidx.c` or `cwptr.c`	`ptaur.c`
		`pcw-m-p.c`
		`pcwd3bar.c`
		`pcwd1d2.c`
		`pcw-knor.c`

References

[Aho90] Aho, A.V. "Algorithms for finding patterns in strings," in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, vol. A* (North-Holland, Amsterdam, 1990) 257–300.

[AC75] Aho, A.V. and M.J. Corasick. "Efficient string matching: an aid to bibliographic search," Comm. ACM, 18(6) (1975) 333–340.

[AMS92] Aoe, J.-I., K. Morimoto, and T. Sato. "An efficient implementation of trie structures," Software — Practice and Experience 22(9), 695–721, Sept. 1992.

[BM77] Boyer, R.S. and J.S. Moore. "A fast string searching algorithm," Comm. ACM, 20(10) (1977) 62–72.

[Com79a] Commentz-Walter, B. "A string matching algorithm fast on the average," in: H.A. Maurer, ed., Proc. 6th Internat. Coll. on Automata, Languages and Programming (Springer, Berlin, 1979) 118–132.

[Com79b] Commentz-Walter, B. "A string matching algorithm fast on the average," Technical report TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.

[Fre60] Fredkin, E. "Trie memory," Comm. ACM, 3(9) (1960) 490–499.

[HU79] Hopcroft, J.E. and J.D. Ullman. *Introduction to Automata, Theory, Languages, and Computation,* Addison-Wesley Publishing Company, Reading, MA, 1979.

[HS91] Hume, S.C. and D. Sunday. "Fast string searching," Software — Practice and Experience, 21(11), 1221–1248, 1991.

[ISO90] ISO/IEC. *Programming languages — C,* International Standard 9899:1990, First edition, 1990-12-15.

[KR88] Kernighan, B.W. and D.M. Ritchie. *The C Programming Language, Second edition,* Prentice-Hall, New Jersey, 1988.

[KMP77] Knuth, D.E., J.H. Morris and V.R. Pratt. "Fast pattern matching in strings," SIAM J. Comput. 6(2) (1977) 323–350.

[Pirk92] Pirklbauer, K. "A study of pattern-matching algorithms," Structured Programming 13, 89–98, 1992.

[PTVF92] Press, W.H., S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing, Second edition,* Cambridge University Press, 1992.

[Smit82] Smit, G. de V. "A comparison of three string matching algorithms," Software — Practice and Experience 12, 57–66, 1982.

[WZ92] Watson, B.W. and G. Zwaan. "A taxonomy of keyword pattern matching algorithms," Computing Science Note 92/27, Eindhoven University of Technology, The Netherlands, 1992.