BIOINFORMATICS ORIGINAL PAPER Vol. 21 no. 10 2005, pages 2225–2229
doi:10.1093/bioinformatics/bti323

*Sequence analysis*

# Using sequence compression to speedup probabilistic profile matching

Valerio Freschi and Alessandro Bogliolo*

Information Science and Technology Institute, University of Urbino, 61029 Urbino, Italy

Received on October 11, 2004; revised on January 13, 2005; accepted on February 9, 2005
Advance Access publication February 15, 2005

**ABSTRACT**

**Motivation:** Matching a biological sequence against a probabilistic pattern (or profile) is a common task in computational biology. A probabilistic profile, represented as a scoring matrix, is more suitable than a deterministic pattern to retain the peculiarities of a given segment of a family of biological sequences. Brute-force algorithms take $O(NP)$ to match a sequence of $N$ characters against a profile of length $P \ll N$.

**Results:** In this work, we exploit string compression techniques to speedup brute-force profile matching. We present two algorithms, based on run-length and LZ78 encodings, that reduce computational complexity by the compression factor of the encoding.

**Contact:** bogliolo@sti.uniurb.it

## 1 INTRODUCTION

Biological sequence analysis is a key task in computational biology aimed at the identification of functional and/or structural relationships between protein or DNA sequences, to infer, for instance, the membership of a newly discovered sequence to a known family. Within a family, conserved domains can be discovered by means of multiple alignments providing alignment blocks collected in public databases (Attwood, 2000; Wu *et al*., 2000). A scoring matrix (also known as profile, position-specific scoring matrix or position-weight matrix) represents a probabilistic pattern of nucleic acids or proteins belonging to a family of related sequences. A conversion method for the computation of a scoring matrix starting from an alignment block is described by Wu *et al*. (1999).

Profiles described by scoring matrices have been increasingly used for sequence analysis for their intrinsic capability of capturing more subtle relationships with respect to deterministic patterns even if enhanced by regular expressions (Attwood, 2000). Matching a sequence against a probabilistic profile (hereafter called probabilistic profile matching) is the core task of packages derived from the extensively used BLAST program (Altschul *et al*., 1997; Schaffer *et al*., 1999). The brute-force algorithm for probabilistic profile matching takes $O((N-P+1)P)$ steps (where $N$ is the length of the sequence and $P$ is the length of the probabilistic profile) to compute position-specific scores according to the scoring matrix representing the profile. Despite advanced algorithms recently proposed to enhance the performance of brute-force computation (Rajasekaran *et al*., 2002; Dorohonceanu and Nevill-Manning, 2000; Wu *et al*., 2000), the complexity of probabilistic profile matching is still regarded as an open issue in computational biology (Gonnet, 2004).

*To whom correspondence should be addressed.

```
max_score = MINSCORE;
p=0;
while (p<N-P+1) {
  score = 0;
  // BEGIN SEGMENTAL SCORE COMPUTATION
  i = 0;
  while (i<P) {
    c = x[p+i];
    score += S[i][c];
    i++;     // increment character counter
  }
  // END SEGMENTAL SCORE COMPUTATION
  if (score > max_score) max_score = score;
  p++;
}
```

**Fig. 1.** Brute-force algorithm.

In this work we introduce two new algorithms that exploit sequence compression techniques (namely, run-length and Lempel–Ziv encodings) to reduce the computational effort of probabilistic profile matching, achieving a speedup proportional to the compression factor.

In the rest of this section we provide a minimum background on probabilistic profile matching and string compression.

### 1.1 Probabilistic profile matching

We denote by $X$ a sequence of $N$ characters taken from an alphabet of $\Sigma$ symbols. A profile of length $P$ is represented by means of scoring matrix $S$ of $P$ rows and $\Sigma$ columns: an entry $S[i][a]$ denotes the score of symbol $a$ when it appears in position $i$ in the profile.

The brute-force algorithm for probabilistic profile matching proceeds by sliding a window of length $P$ over string $X$. For each position of the window, a segmental score is computed as the sum of the scores assigned to $P$ characters within the window according to $S$. The result of the matching are the maximum segmental scores computed over all possible positions of the sliding window. Since the window may take $N-P+1$ different positions and for each position a segmental score is computed in $O(P)$ steps, the brute-force algorithm has complexity $O((N-P+1)P) \simeq O(NP)$.

The pseudocode of the algorithm is reported in Figure 1. Consider, for instance, a profile of length 4 to be matched to string $X = ataatcaactcg$ of 12 characters. The algorithm takes $(12-4+1)\cdot 4 = 36$ steps (pictorially represented in Fig. 3a) to complete computation.

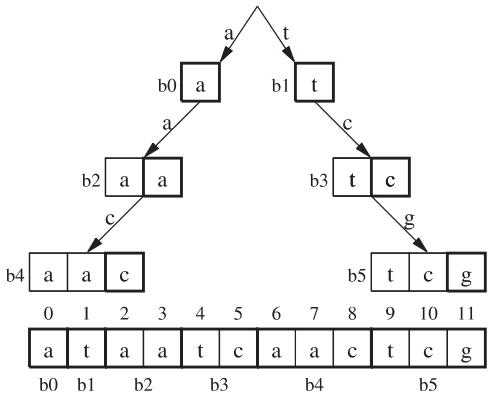


**Fig. 2.** The LZ78-tree encoding a short string of 12 characters.

Several approaches have been proposed to improve the performance of brute-force probabilistic profile matching. Wu *et al*. (2000) developed a branch-and-bound strategy called lookahead scoring. When computing a segmental score, partial scores are compared with partial thresholds, derived from the maximum possible score associated with the last (not yet matched) part of the profile. If the partial score is below the threshold, segmental score computation can be safely interrupted and restarted at a new position of the sliding window. The speedup achieved depends on the effectiveness of the partial thresholds that depend on the probabilistic profile. Permutations can be applied to further improve performance. The residues in positions statistically providing the highest contribution to the segmental score are considered first in order to reduce the time taken to possibly decide to reject the score.

Suffix trees (Gusfield, 1997) can be used as data structures to provide more pruning opportunities (Dorohonceanu and Nevill-Manning, 2000). Segments that share the first characters are associated with paths in the tree that share the first edges from the root. Segmental scores are computed during a depth-first traversal of the tree. As soon as a segmental score is found to be below a given acceptance threshold, all segments represented by the leaves of the subtree rooted in the current node can be discarded without further computation. Similar results can be achieved using suffix arrays (Beckstette *et al*., 2004).

It is worth noting, however, that lookahead scoring and suffix trees/arrays trade off sensitivity for performance. In fact, they provide a speedup dependent on the acceptance threshold set by the user. No information is provided about the segmental scores that fall below the acceptance threshold. The brute-force algorithm, in contrast, returns segmental scores associated with each position of the pattern on the target string.

Rajasekaran *et al*. (2002) proposed a different algorithm, based on fast Fourier transform (FFT), that levers the properties of cyclic convolution to achieve $O((N+P-1)\log(N+P-1))$ complexity. The actual advantage of the FFT-based algorithm with respect to brute-force depends on the values of $N$ and $P$, and it can be counterproductive when $P < \log N$.

### 1.2 String compression

String compression has been exploited not only to reduce storage and communication needs but also to improve the complexity of string processing algorithms. In particular, compressed pattern matching (exact or approximate) has been widely investigated in the fields of information retrieval and computational biology (Amir *et al*., 1996; Crochemore *et al*., 2003; Karkkainen *et al*., 2000).

Run-length encoding (RLE) (Sayood, 2000) and Lempel–Ziv compression (LZ78) (Ziv and Lempel, 1978) are among the most common string compression techniques. RLE encodes $l$ consecutive occurrences of the same symbol $a$ (i.e. a run) as a couple $(l, a)$. LZ78 exploits repetitions of substrings to represent a string as a sequence of incremental blocks. Each block is composed of two elements: a pointer to an already encountered substring (if any) and a character to be appended to that known substring to complete the block. Since each block is defined as an extension of a previous one, all the blocks of the LZ78 encoding of a given string can be represented in a tree, as shown in Figure 2 for a string of 12 characters. The tree is incrementally constructed in linear time during the parsing of the string. The first character of the string is a block by itself that is added to the tree as a child of an empty root. The edge leading to each block is labeled with the last character of the block. During the parsing of the string the tree is traversed from the root, following a path labeled by the characters read on the string. When the path cannot be extended any more, a new branch is created leading to a new block that extends one character of the block associated with the parent node. Each time a new block is added the traversal of the tree restarts from the root. Assume, for instance, that only the first six characters of the string of Figure 2 have been parsed so that only the first four blocks have been added to the LZ78-tree. Starting from the root of the tree, characters 6 and 7 of the string lead to the node associated with block b2. The next character is a 'c', that does not correspond to any path. Hence, a new leaf is added to the tree corresponding to block b4 = 'aac', that extends block b2. The complexity of the representation is proportional to the number of different blocks (i.e. the number of nodes in the tree).

The compression factor (i.e. the normalized size of the compressed representation) achieved by RLE and LZ78 on a string $X$ of $N$ characters can be expressed as $1/l_{\text{avg}}$ and $1/\log(N)$, respectively, where $l_{\text{avg}}$ is the average length of the runs of $X$, and $\log(N)$ has been shown to be the asymptotic average length of LZ78 blocks (Ziv and Lempel, 1978).

## 2 ALGORITHM

In this work we present two algorithms for probabilistic profile matching that exploit either RLE or LZ78 compression of string $X$. The proposed algorithms reduce the complexity of the brute-force computation by the compression factor achieved by the corresponding encodings on $X$.

### 2.1 Exploiting run length encoding

We define a telescopic scoring matrix (denoted by *S_RLE*) that can be easily obtained (in a preprocessing step) from the scoring matrix ($S$) of the profile of interest:

$$S\_RLE[i][a] = \sum_{j=i}^{P-1} S[j][a] \tag{1}$$

As for the brute-force algorithm, a sliding window of size $P$ is shifted one character at the time along sequence $X$. Matrix *S_RLE* is used instead of $S$ to lever RLE to speedup the computation of the segmental

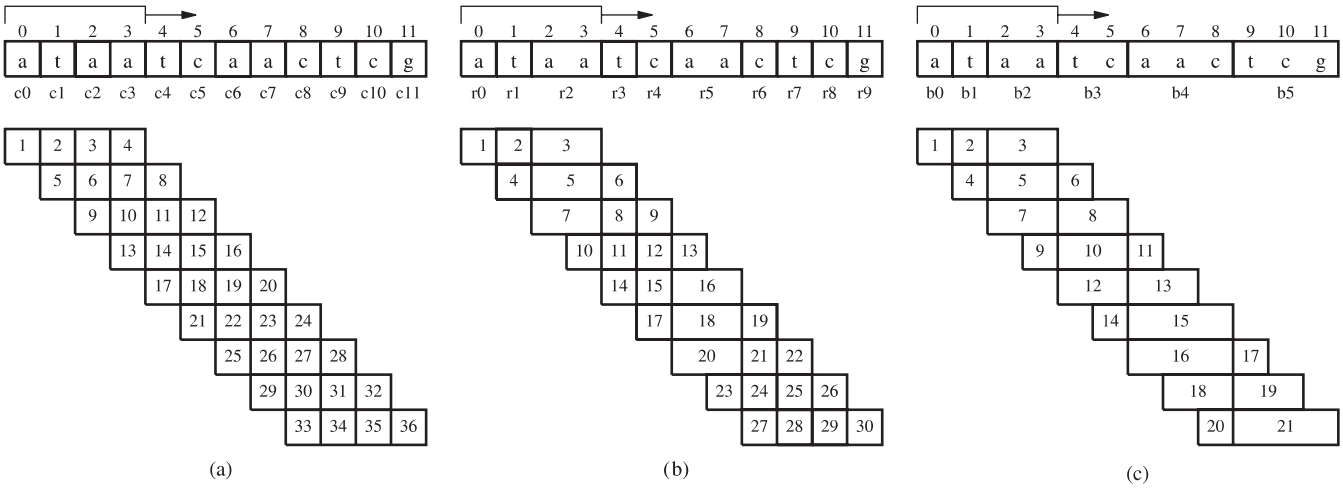


**Fig. 3.** Graphical representation of the steps required to perform probabilistic profile matching by means of **(a)** brute-force algorithm, **(b)** RLE algorithm and **(c)** LZ78 algorithm.

scores. Consider a run $(l, a)$ located in the window from character $i$ to character $i + l - 1$. The contribution of the run to the score can be computed in $O(1)$, independently of the length of the run, as $S\_RLE[i][a] - S\_RLE[i + l][a]$.

The pseudocode of the algorithm is shown in Figure 4. To match string $X = ataatcaactcg = (1, a)(1, t)(2, a)(1, t)(1, c)(2, a)(1, c)(1, t)(1, c)(1, g)$ against a profile of size 4 it takes 30 steps instead of 36 (Fig. 3b).

The RLE algorithm levers compression by allowing the direct $O(1)$ computation of a run (i.e. the computation of the contribution of all its characters to the segmental score). If we denote by $l_{\text{avg}}$ the average length of the runs of the sequence to be matched, in a window of $P$ characters we find on average $P/l_{\text{avg}}$ runs, so that the computational complexity of each window is $O(P/l_{\text{avg}})$. Since the window segmental score must be evaluated at all possible $N - P + 1$ positions, the complexity of the algorithm that exploits RLE can be expressed as $O((1/l_{\text{avg}})(N - P + 1)P)$.

## 2.2 Exploiting LZ78 encoding

In order to exploit LZ78 compression to speedup probabilistic profile matching we need to be able to compute in constant time the contribution of each block of the LZ78 representation to the segmental scores. To this purpose, we associate to each node of the LZ78 tree (i.e. to each block) two data structures: an integer number representing the length of the block and an array of $P + l_{\max} - 1$ elements representing the contribution of the block to the segmental score for each possible position of the block within the sliding window ($l_{\max}$ is the maximum size of a block, corresponding to the depth of the LZ78 tree). The incremental nature of LZ78 encoding is exploited to incrementally update the arrays of scores during the parsing of the compressed string.

The pseudocode of the algorithm is shown in Figure 5. At each iteration of the inner loop, a block is processed in constant time to: (1) lookup the data structure of its parent node to obtain a precomputed position-specific score, (2) add the contribution of the last character of the block and (3) update the data structure associated with the block.

```
max_score = MINSCORE;
p = 0; r = 0; p0 = 0;
while (p<N-P+1) {
  score = 0;
  // BEGIN SEGMENTAL SCORE COMPUTATION
  i = 0;
  start = 0;
  stop = 0;
  while (start<P) {
    c = x[r+i]->c;
    length = x[r+i]->l;
    stop = start+length;
    if (i=0) stop -= p0;
                  // first run of the pattern
    score += S_RLE[start][c];
    if (stop<P)
      score -= S_RLE[stop][c];
    start += length;
    i++;
                // increment run counter
  }
  // END SEGMENTAL SCORE COMPUTATION
  if (score > max_score) max_score = score;
  p ++;
  p0 ++;
  if (p0 >= x[r]->l) {
    r ++;
    p0 = 0;
  }
}
```

**Fig. 4.** The pseudocode of RLE algorithm.

Figure 3c represents the computation steps of the proposed algorithm. Matching our example string $X$ against a profile of length $P = 4$ takes 21 steps.

The computational complexity can be evaluated starting from considerations similar to those made in the previous section for RLE.

```
max_score = MINSCORE;
p = 0; r = 0; p0 = 0;
while (p<N-P+1) {
  score = 0;
  // BEGIN SEGMENTAL SCORE COMPUTATION
  i = 0;
  start = -p0;
  while (start<P-1) {
    c = x[r+i]->newc;
    length = x[r+i]->l;
    prev = x[r+i]->prev;
    x[r+i]->S[start] = prev->S[start];
    if (start+length-1 < P)
      x[r+i]->S[start]+=S[start+length-1][c];
    score += x[r+i]->S[start];
    start += length;
    i++;        // increment block counter
  }
  // END SEGMENTAL SCORE COMPUTATION
  if (score > max_score) max_score = score;
  p ++;
  p0 ++;
  if (p0 >= x[r]->l) {
    r ++;
    p0 = 0;
  }
}
```

**Fig. 5.** The pseudocode of LZ78 algorithm.

The asymptotic average length of blocks obtained by performing the LZ78-parsing of a sequence of length $N$ has been demonstrated to be $\log(N)$ (Ziv and Lempel, 1978). Since the proposed algorithm computes in constant time the contribution of each block to the segmental score, the computation of a window of $P$ characters takes $O(P/\log(N))$ steps, leading to an overall computation of $O((1/\log(N))(N-P+1)P)$.

## 3 RESULTS AND DISCUSSION

In this work we have introduced two new algorithms for the computation of probabilistic profile matching exploiting sequence compression, either RLE or LZ78. The proposed algorithms compute all position-specific segmental scores with a time complexity reduced (with respect to the brute-force algorithm) by the compression factor achieved through the encoding. In particular, the asymptotic complexity can be written as $O((1/l_{\text{avg}})(N-P+1)P)$ for RLE compression and $O((1/\log N)(N-P+1)P)$ for LZ78 factorization (where $N$ is the length of the uncompressed string $X$, $P$ is the length of the profile and $l_{\text{avg}}$ is the average run length of $X$). When $N \gg P$, the complexity reduces to $O((N/l_{\text{avg}})P)$ and $O((N/\log N)P)$, respectively.

Since the improvement provided by the proposed approaches depends on the efficiency of the encodings, Figure 6 shows the number of runs and LZ78 blocks for DNA and protein sequences of different lengths taken from the human genome (http://www.ncbi.nlm.nih.gov/). Sequences have been randomly chosen from chromosomes 1, 2 and 3 of the human genome and then concatenated to obtain single DNA/protein sequences of 45 000 nucleotides/aminoacids. The graphs refer to the progressive encoding of

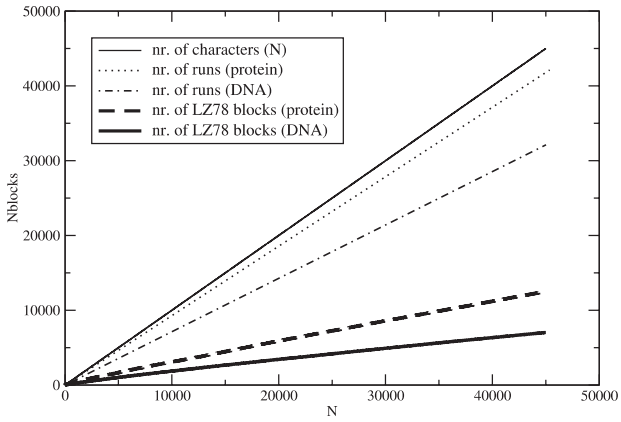


**Fig. 6.** Number of runs and LZ78 blocks in the compressed representation of DNA and protein sequences, plotted as functions of the number of characters in the uncompressed strings.

these benchmark sequences. For each value of $N$ the efficiency of the encodings was evaluated on the first $N$ characters of the benchmark sequences.

While RLE provides little improvement with respect to brute-force algorithm (the number of runs is close to the number of characters of the original sequences), LZ78 provides a sizeable advantage. The advantage is greater for DNA sequences, since the lower the number of symbols in the alphabet ($|\Sigma|$) the higher the efficiency of LZ78. In fact, the average number of blocks can be expressed as $N/\log_{|\Sigma|}(N) = \log(|\Sigma|)N/\log(N)$.

Figure 7 provides a comparison between the asymptotic complexity of the proposed algorithm based on LZ78 and the algorithm based on FFT (Rajasekaran *et al.*, 2002). The complexity ratio between FFT and LZ78 algorithms is plotted for different representative values of $P$ and $N$ for both DNA and protein sequences (our algorithm is more efficient whenever the complexity ratio is $>1$). When dealing with proteins, the FFT-based algorithm provides better performance when matching a long profile against a short sequence. When dealing with DNA sequences, the proposed LZ78-based algorithm is more efficient than FFT for all configurations of $P$ and $N$ shown in the graph.

As a final remark, we observe that heuristic branch-and-bound (B&B) approaches (Wu *et al.*, 2000; Beckstette *et al.*, 2004; Dorohonceanu and Nevill-Manning, 2000) cannot be directly compared with the proposed algorithms since the former trade off sensitivity for performance, while the latter do not, providing exactly the same results provided by the brute-force algorithm. On the other hand, B&B techniques can be easily combined with the RLE-based algorithm to further reduce computational complexity. In fact, B&B is used to safely abandon segmental score computation whenever the partial score is recognized to be below a precomputed position-dependent threshold. Our algorithm speeds up segmental score computation by handling a run at the time instead of a single character. B&B techniques can be applied to the RLE-based algorithm by comparing partial segmental scores with position-dependent thresholds whenever the contribution of a new run is added. In contrast, B&B cannot be directly applied to the LZ78-based algorithm since the $O(1)$ computation of the contribution of each block is based on the precomputed score of the block associated with its parent node

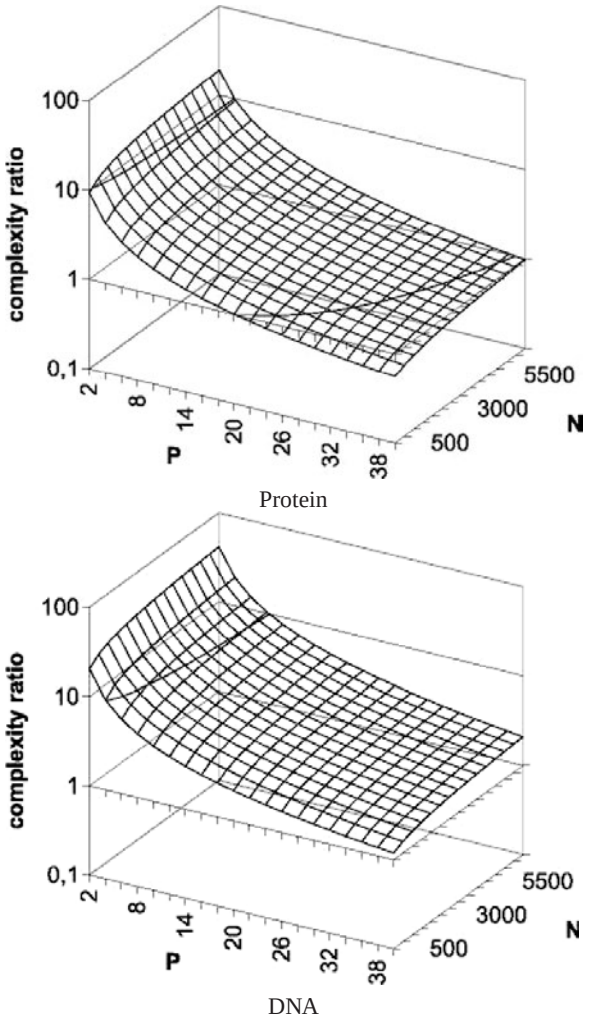


**Fig. 7.** Complexity ratio of FFT-based to LZ78-based probabilistic profile matching algorithms, used to match a profile of length $P$ against a protein/DNA sequence of length $N$. When the ratio is $>1$, the LZ78-based algorithm is more efficient than the FFT-based one.

in the LZ78 tree. Hence, the scores of all blocks in the incremental computation chain need to be computed regardless of B&B conditions.

### 3.1 Implementation issues

The discussion conducted so far was based on asymptotic complexities. The practical advantage of the proposed techniques depends, however, on the multiplicative constants that represent the average execution time of the inner loop of the algorithms. On the other hand, the execution time of an algorithm is strongly dependent on its implementation. In this section we do not provide an implementation-dependent comparison of performance. Rather, we discuss the implementation issues that may impact the execution time of the inner loops of each algorithm and we refer to the comparison of their asymptotic complexities to evaluate the performance margins. For instance, from Figure 6 we observe that LZ78 provides an asymptotic speedup of about five times over brute-force algorithm when dealing with proteins of 20 000 residues. This means that the LZ78 algorithm provides a practical speedup as long as the execution of its inner loop takes less than five times that of the brute-force algorithm. Similarly, the complexity ratio of Figure 7 shows that LZ78 profile matching is more efficient than FFT-based matching when searching, for instance, a profile of $P = 10$ characters in a protein of $N = 500$ residues, even if its inner loop is five times slower.

The additional algorithmic complexity of the inner loops of RLE and LZ78 with respect to brute-force algorithm can be evaluated from Figures 1, 4 and 5.

The RLE algorithm requires additional arithmetic operations to compute run contributions as the difference between two precomputed scores. Since the margins provided by RLE compression on biological sequences is quite small (Fig. 6), a straightforward software implementation of the RLE algorithm may be counterproductive, but score differences could be precomputed and stored in a three dimensional scoring matrix to speedup execution.

The inner loop of the LZ78 algorithm requires two additions to evaluate the position-dependent contribution of each block: one for adding the contribution of the parent node and one for adding the contribution of the new character. Moreover, the data structure used to store block scores is much larger than a simple scoring matrix and structured as a tree. If it exceeds the size of the first-level cache, miss penalties can significantly slow down execution. The memory required by LZ78 is $O(N)$. On the other hand, LZ78 compression provides wide margins to compensate the overhead. It is also worth noting that the speedup achieved by LZ78 grows with $O(\log N)$, so that any constant overhead could in principle be overcome for long enough sequences.

## REFERENCES


Altschul,S.F. *et al*. (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, **25**, 3389–3402.

Amir,A. *et al*. (1996) Let sleeping files lie: pattern matching in Z-compressed files. *J. Comput. Syst. Sci.*, **52**, 299–307.

Attwood,T.K. (2000) The role of pattern databases in sequence analysis. *Brief. Bioinform.*, **1**, 45–59.

Beckstette,M. *et al*. (2004) PoSSuMsearch: fast and sensitive matching of postion specific scoring matrices using enhanced suffix arrays. In *Proceedings of the German Conference on Bioinformatics 2004*. Bielefeld, Germany, October 4–6.

Crochemore,M. *et al*. (2003) A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput.*, **32**, 1654–1673.

Dorohonceanu,B. and Nevill-Manning,C.G. (2000) Accelerating protein classification using suffix trees. In *Proceedings of 8th International Conference on Intelligent Systems for Molecular Biology 2000*, pp. 128–133.

Gonnet,G.H. (2004) Some string matching problems from bioinformatics which still need better solutions. *J. Discr. Algorithms*, **2**, 3–15.

Gusfield,D. (1997) *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge.

Karkkainen,J. *et al*. (2000) Approximate string matching over Ziv–Lempel compressed text. *Proceedings of 11 CPM*, Lecture Notes in Computer Science Vol. 1848, pp. 195–209.

Rajasekaran,S. *et al*. (2002) The efficient computation of position-specific match scores with the fast Fourier transform. *J. Comput. Biol.*, **9**, 23–33.

Sayood,K. (2000) *Introduction to Data Compression*, 2nd edn. Morgan Kaufmann, San Francisco, CA.

Schaffer,A.A. *et al*. (1999) IMPALA: matching a protein sequence against a collection of PSI-BLAST-constructed position-specific score matrices. *Bioinformatics*, **15**, 1000–1011.

Wu,T.D. *et al*. (1999) Minimal risk scoring matrices for sequence analysis. *J. Comput. Biol.*, **6**, 219–235.

Wu,T.D. *et al*. (2000) Fast probabilistic analysis of sequence function using scoring matrices. *Bioinformatics*, **16**, 233–244.

Ziv,J. and Lempel,A. (1978) Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, **24**, 530–536.