Approximate String Matching in Musical Sequences

Maxime Crochemore, Costas S. Iliopoulos, Thierry Lecroq, Yoan J. Pinzon

Approximate String Matching in Musical Sequences*

Maxime Crochemore[1], Costas S. Iliopoulos[2 †],
Thierry Lecroq[3] and Y. J. Pinzon[2 ‡]

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, France.
`mac@univ-mlv.fr`,
[2] Dept. of Computer Science, King's College London, London WC2R 2LS, England, and School of Computing, Curtin University of Technology, GPO Box 1987 U, WA. Australia
`{csi,pinzon}@dcs.kcl.ac.uk`,
[3] LIFAR - ABISS, Université de Rouen, 76821 Mont Saint Aignan Cedex, France.
`lecroq@dir.univ-rouen.fr`

Abstract. Here we consider computational problems on δ-approximate and (δ, γ)-approximate string matching. These are two new notions of approximate matching that arise naturally in applications of computer assisted music analysis. We present fast, efficient and practical algorithms for these two notions of approximate string matching.

Key words: String algorithms, approximate string matching, dynamic programming, computer-assisted music analysis.

1 Introduction

This paper focuses on a set of string pattern-matching problems that arise in musical analysis, and especially in musical information retrieval. A musical score can be viewed as a string: at a very rudimentary level, the alphabet could simply be the set of notes in the chromatic or diatonic notation, or the set of intervals that appear between notes (e.g. pitch may be represented as MIDI numbers and pitch intervals as number of semitones). Approximate repetitions in one or more musical works play a crucial role in discovering similarities between different musical entities and may be used for establishing "characteristic signatures" (see [6]). Such algorithms can be particularly useful for melody identification and musical retrieval.

The approximate repetition problem has been extensively studied over the last few years. Efficient algorithms for computing the approximate repetitions are directly applicable to molecular biology (see [7, 9, 12]) and in particular in DNA sequencing by

*This work was partially supported by a NATO grant PST.CLG.977017.
†Partially supported by a Marie Curie fellowship, Wellcome and Royal Society grants.
‡Partially supported by an ORS studentship and EPSRC Project GR/L92150.

hybridization ([13]), reconstruction of DNA sequences from known DNA fragments (see [15, 16]), in human organ and bone marrow transplantation as well as the determination of evolutionary trees among distinct species ([15]).

The approximate matching problem has been used for a variety of musical applications (see overviews in McGettrick [11]; Crawford et al [6]; Rolland et al [14]; Cambouropoulos et al [3]). It is known that exact matching cannot be used to find occurrences of a particular melody. Approximate matching should be used in order to allow the presence of errors. The number of errors allowed will be referred to as δ. This paper focuses in one special type of approximation that arise especially in musical information retrieval, i.e. δ-approximation. Most computer-aided musical applications adopt an absolute numeric pitch representation (most commonly MIDI pitch and pitch intervals in semitones; duration is also encoded in a numeric form). The absolute pitch encoding, however, may be insufficient for applications in tonal music as it disregards tonal qualities of pitches and pitch-intervals (e.g. a tonal transposition from a major to a minor key results in a different encoding of the musical passage and thus exact matching cannot detect the similarity between the two passages). One way to account for similarity between closely related but non-identical musical strings is to use what will be referred to as δ-approximate matching (and γ-approximate matching). In δ-approximate matching, equal-length patterns consisting of integers match if each corresponding integer differs by not more than δ- e.g. a C-major $\{60, 64, 65, 67\}$ and a C-minor $\{60, 63, 65, 67\}$ sequence can be matched if a tolerance $\delta = 1$ is allowed in the matching process (γ-approximate matching is described in the next section).

In [4], a number of efficient algorithms for δ-approximate matching were presented (i.e. the Shift-And algorithm and Shift-Plus algorithm). The Shift-And algorithm is based on the $O(1)$-time computation of different states for each symbol in the text. Hence the overall complexity is $O(n)$. These algorithms use the bitwise technique. It is possible to adapt fast and practical exact pattern matching algorithms to these kind of approximations. In this paper we will present the adaptations of the Tuned-Boyer-Moore [8], the Skip-Search algorithm [5] and the Maximal-Shift algorithm [17] and present some experiments to assert that these adaptations are faster than the algorithms using the bitwise technique.

The paper is organised as follows. In the next section we present some basic definitions for strings and background notions for approximate matching. In Sections 3-5 we present the adaptation of Tuned-Boyer-Moore, Skip-Search and Maximal-Shift algorithms to speed-up δ-approximate pattern matching algorithms and in section 6 to speed-up (δ, γ)-approximate pattern matching algorithms. In section 7 we present the experimental results of these algorithms. Finally in Section 8 we present our conclusions.

2 Background and basic string definitions

A *string* is a sequence of zero or more symbols from an alphabet Σ; the string with zero symbols is denoted by ϵ. The set of all strings over the alphabet Σ is denoted by Σ^*. A string x of length n is represented by $x_1 \dots x_n$, where $x_i \in \Sigma$ for $1 \leq i \leq n$. A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$; we equivalently say that the string w occurs at position $|u| + 1$ of the string x. The position $|u| + 1$ is said to be the *starting position* of w in x and the position $|w|+|u|$ the *end position* of w in x. A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$. Similarly, w is a *suffix* of x if $x = uw$ for $u \in \Sigma^*$.

The string xy is a *concatenation* of two strings x and y. The concatenations of k copies of x is denoted by x^k. For two strings $x = x_1 \dots x_n$ and $y = y_1 \dots y_m$ such that $x_{n-i+1} \dots x_n = y_1 \dots y_i$ for some $i \geq 1$, the string $x_1 \dots x_n y_{i+1} \dots y_m$ is a *superposition* of x and y. We say that x and y *overlap*.

Let x be a string of length n. The integer p is said to be a *period* of x, if $x_i = x_{i+p}$ for all $1 \leq i \leq n-p$. *The period* of a string x is the smallest period of x. A string y is a *border* of x if y is a prefix and a suffix of x.

Let Σ be an alphabet of integers and δ an integer. Two symbols a, b of Σ are said to be δ-approximate, denoted $a \stackrel{\delta}{=} b$ if and only if

$$|a-b| \leq \delta$$

We say that two strings x, y are δ-approximate, denoted $x \stackrel{\delta}{=} y$ if and only if

$$|x| = |y|, \text{ and } x_i \stackrel{\delta}{=} y_i, \ \forall i \in \{1, \dots, |x|\} \tag{2.1}$$

For a given integer γ we say that two strings x, y are γ-approximate, denoted $x \stackrel{\gamma}{=} y$ if and only if

$$|x| = |y|, \text{ and } \sum_{i=1}^{|x|} |x_i - y_i| \leq \gamma \tag{2.2}$$

Furthermore, we say that two strings x, y are $\{\gamma, \delta\}$-approximate, denoted $x \stackrel{\delta,\gamma}{=} y$, if and only if x and y satisfy conditions (2.1) and (2.2).

3 δ-Tuned-Boyer-Moore Approximate Pattern Matching

The problem of *δ-approximate pattern matching* is formally defined as follows: given a string $t = t_1 \dots t_n$ and a pattern $p = p_1 \dots p_m$ compute all positions j of t such that

$$p \stackrel{\delta}{=} t[j..j+m-1]$$

A naive solution of this problem is to build an Aho-Corasick automaton (see [1]) of all strings that are δ-approximate to p and then use the automaton to process t. The time required to build the automaton is $O(|\Sigma|^\delta)$, thus this method is of no practical use as e.g we can have $|\Sigma| \approx 180$ and $|\delta| \approx 10$. In [4] an efficient algorithm was presented based on the $O(1)$-time computation of the "delta states" by using bit operations under the assumption that $m \leq w$, where w is the number of bits in a machine word.

Here we present an adaptation of the Tuned-Boyer-Moore for exact pattern matching algorithm to δ-approximate pattern matching. The exact pattern matching problem consists in finding one or more (generally all) exact occurrences of a pattern p of length m in a text t of length n. Basically a pattern matching algorithm uses a window which size is equal to the length of the pattern. It first aligns the left ends of the window and the text. Then it checks if the pattern occurs in the window and shifts the window to the right. It repeats the same procedure again until the right end of the window goes beyond the right end of the text.

The TUNED-BOYER-MOORE algorithm is a very fast practical variant of the famous BOYER-MOORE algorithm [2]. It only uses the occurrence shift function to perform the shifts. The occurrence shift function is defined for each symbol a in the alphabet Σ as follows:

$$\mathit{shift}[a] = \min\{\{m - i \mid p_i = a\} \cup \{m\}\}$$

The TUNED-BOYER-MOORE algorithm gains its efficiency by unrolling three shifts in a very fast skip loop to locate the occurrences of the rightmost symbol of the pattern in the text. Once an occurrence of p_m is found, it checks naively if the whole pattern occurs in the text. Then the shift consists in aligning the rightmost symbol of the window with the rightmost reoccurrrence of p_m in $p_1 \dots p_{m-1}$, if any. The length s of this shift is defined as follows:

$$s = \min\{\{m - i \mid p_i = p_m \text{ and } i > 0\} \cup \{m\}\}$$

To do δ-approximate pattern matching, the shift function can be defined to be for each symbol a in the alphabet Σ the distance from the right end of the pattern of the closest symbol p_i such that $p_i \stackrel{\delta}{=} a$:

$$\mathit{shift}[a] = \min\{\{m - i \mid p_i \stackrel{\delta}{=} a\} \cup \{m\}\}$$

Then the length of the shift s becomes:

$$s = \min\{\{m - i \mid p_i \stackrel{2\delta}{=} p_m \text{ and } i > 0\} \cup \{m\}\}$$

The pseudo-code for δ-TUNED-BOYER-MOORE algorithm can be found in Figure 1.

4 δ-SKIP-SEARCH Approximate Pattern Matching

In the SKIP-SEARCH algorithm, for each symbol of the alphabet, a bucket collects all of that symbol's positions in p. When a symbol occurs k times in the pattern, there are k corresponding positions in the symbol's bucket. When the word is much shorter than the alphabet, many buckets are empty. The buckets are stored in a table z defined as follows:

$$z[a] = \{i \mid p_i = a\}$$

The main loop of the search phase consists of examining every mth text symbol, t_j (so there will be n/m main iterations). Then for t_j, it uses each position in the bucket $z[t_j]$ to obtain a possible starting point of p in t and checks if the pattern occurs at that position.

To do δ-approximate pattern matching, the buckets can be computed as follows:

$$z[a] = \{i \mid p_i \stackrel{\delta}{=} a\}$$

Figure 2 shows the pseudo-code for δ-SKIP-SEARCH algorithm.

```
δ-TUNED-BOYER-MOORE(p, m, t, n, δ)
▷ Preprocessing
for all a ∈ Σ
    do shift[a] ← min{{m − i | p_i =^δ a} ∪ {m}}
s ← min{{m − i | p_i =^{2δ} p_m} ∪ {m}}
t_n ... t_{n+m−1} ← (p_m)^m
▷ Searching
j ← m
while j ≤ n
    do k ← shift[t_j]
       while k ≠ 0
           do j ← j + k
              k ← shift[t_j]
              j ← j + k
              k ← shift[t_j]
              j ← j + k
              k ← shift[t_j]
       if p_1 ... p_{m−1} =^δ t_{j−m+1} ... t_{j−1} and j ≤ n
         then REPORT(j − m + 1)
       j ← j + s
```

Figure 1: Adaptation of the TUNED-BOYER-MOORE exact pattern matching algorithm to do δ-approximate pattern matching.

5 δ-MAXIMAL-SHIFT Approximate Pattern Matching

Sunday [17] designed an exact string matching algorithm where the pattern positions are scanned from the one which will lead to a larger shift to the one which will lead to a shorter shift, in case of a mismatch. Doing so one may hope to maximize the lengths of the shifts and thus to minimize the overall number of comparisons.

Formally we define a permutation

$$\sigma : \{1, 2, \ldots, m, m+1\} \to \{1, 2, \ldots, m, m+1\}$$

and a function *shift* such that

$$\mathit{shift}[\sigma(i)] \stackrel{\gamma}{=} \mathit{shift}[\sigma(i+1)]$$

for $1 \le i < m$ and

$$\mathit{shift}[\sigma(i)] = \min\{\ell \mid \text{ for } 1 \le j < i, p_{\sigma(j)-\ell} = p_{\sigma(j)} \text{ and } p_{\sigma(i)-\ell} \neq p_{\sigma(i)}\}$$

for $1 \le i \le m$ and $\sigma(m+1) = m+1$. Furthermore *shift*$[m+1]$ is set with the value of the period of the pattern p.

We also define a function *bc* for each symbol of the alphabet:

```
δ-SKIP-SEARCH(p, m, t, n, δ)
▷ Preprocessing
for all a ∈ Σ
    do z[a] ← {i | p_i =^δ a}
▷ Searching
j ← m
while j ≤ n
    do for all i ∈ z[t_j]
            do if p =^δ t_{j-i} ... t_{j-i+m-1}
                  then REPORT(j − i)
       j ← j + m
```

Figure 2: Adaptation of the SKIP-SEARCH exact pattern matching algorithm to do δ-approximate pattern matching.

$$bc[a] = \begin{cases} \min\{j \mid 0 \leq j < m \text{ and } p_{m-j} = a\} & \text{, if } a \text{ occurs in } p \\ m & \text{, otherwise} \end{cases}$$

for $a \in \Sigma$.

Then, when the pattern is aligned with the $t[j..j+m-1]$ the comparisons are performed in the following order $\sigma(1), \sigma(2), \ldots, \sigma(m)$ until the whole pattern is scanned or a mismatch is found. If a mismatch is found when comparing $p[\sigma(i)]$ then a shift of length $\max\{\mathit{shift}[\sigma(i)], bc[t[j+m+1]]\}$ is performed. Otherwise an occurrence of the pattern is found and the length of the shift is equal to the maximum value between the period of the pattern and $bc[t[j+m+1]]$. Then the comparisons resume with $p_{\sigma(1)}$ without keeping any memory of the comparisons previously done.

To perform δ-approximate string matching the two functions can be redefined as follows:

$$\mathit{shift}[\sigma(i)] = \min\{\ell \mid \text{ for } 1 \leq j < i, p_{\sigma(j)-\ell} =_{2\delta} p_{\sigma(j)} \text{ and } p_{\sigma(i)-\ell} \neq_{\delta} p_{\sigma(i)}\}$$

for $1 \leq i \leq m$ and

$$\mathit{shift}[m+1] = \min\{\ell \mid p[i] =_{2\delta} p[i+\ell] \text{ for } 1 \leq i \leq m-\ell\}$$

and

$$bc[a] = \begin{cases} \min\{j \mid 0 \leq j < m \text{ and } p_{m-j} =_{\delta} a\} & \text{, if such a } j \text{ exists} \\ m & \text{, otherwise} \end{cases}$$

for $a \in \Sigma$.

The preprocessing phase can be done in $O(m^2)$. Figure 3 gives the pseudo-code of the searching phase.

```
δ-MAXIMAL-SHIFT(p, m, t, n, δ)
▷ Searching
j ← 0
while j ≤ n − m
    do i ← 1
        while i ≤ m and p[σ(i)] = t[j + σ(i)]
            do i ← i + 1
        if i > m
          then REPORT(j)
        j ← j + max{shift[σ(i)], bc[t[j + m + 1]]}
```

Figure 3: Adaptation of the MAXIMAL-SHIFT exact pattern matching algorithm to do δ-approximate pattern matching.

6 (δ, γ)-Approximate String Matching Algorithms

The problem of (δ, γ)-*approximate pattern matching* is formally defined as follows: given a string $t = t_1 \dots t_n$ and a pattern $p = p_1 \dots p_m$ compute all positions j of t such that

$$p \stackrel{\delta,\gamma}{=} t[j..j + m - 1]$$

In [4] this problem was solved by making use of the SHIFT-AND algorithm to find the δ-approximate matches of the pattern p in t. Once a δ-approximate match was found, it was then tested to check whether it is also a γ-approximate match. This was done by computing successive “delta states” and “gamma states” in $O(1)$ time using bit operations under the assumption that $m \leq w$ where w is the number of bits in a machine word.

In order to adapt the δ-TUNED-BOYER-MOORE, δ-SKIP-SEARCH and δ-MAXIMAL-SHIFT algorithms to the case of (δ, γ)-approximation, it just suffices to adapt the naive check of the pattern. The resulting algorithms are named (δ, γ)-TUNED-BOYER-MOORE algorithm, (δ, γ)-SKIP-SEARCH algorithm and (δ, γ)-MAXIMAL-SHIFT algorithm.

7 Experimental results

We implemented in C, in a homogeneous way, the following algorithms: SHIFT-AND, δ-TUNED-BOYER-MOORE, δ-SKIP-SEARCH, δ-MAXIMAL-SHIFT, SHIFT-PLUS, (δ, γ)-TUNED-BOYER-MOORE, (δ, γ)-SKIP-SEARCH and (δ, γ)-MAXIMAL-SHIFT.

We randomly built a text of 500k symbols on an alphabet of size $|\Sigma| = 70$. We then searched for each values 100 patterns and took the average running time. Times are measured in hundredth of seconds and include both preprocessing and searching times.

The results for δ-approximation are shown in tables 1 to 5. For the values used in these experiments, the δ-TUNED-BOYER-MOORE algorithm is always faster than the δ-SKIP-SEARCH algorithm which is itself always faster than the SHIFT-AND algorithm.

m	Shift-And	δ-Tuned-Boyer-Moore	δ-Skip-Search
8	32.98	10.78	18.61
9	32.90	10.55	18.11
10	32.93	10.10	17.65
20	32.86	9.32	15.81

Table 1: Running times for δ-approximation with $\delta = 5$.

m	Shift-And	δ-Tuned-Boyer-Moore	δ-Skip-Search
8	33.07	13.40	21.66
9	32.90	13.00	20.94
10	32.93	12.64	20.49
20	32.92	11.97	18.81

Table 2: Running times for δ-approximation with $\delta = 6$.

m	Shift-And	δ-Tuned-Boyer-Moore	δ-Skip-Search
8	33.65	16.65	24.99
9	33.14	16.05	24.06
10	33.05	15.71	23.62
20	32.93	14.82	21.42

Table 3: Running times for δ-approximation with $\delta = 7$.

m	Shift-And	δ-Tuned-Boyer-Moore	δ-Skip-Search
8	34.72	21.18	29.15
9	33.41	20.03	27.64
10	33.07	19.12	26.85
20	32.81	18.20	24.41

Table 4: Running times for δ-approximation with $\delta = 8$.

The results for (δ, γ)-approximation are shown in tables 6 to 10. For the values used in these experiments, the (δ, γ)-Tuned-Boyer-Moore algorithm is always faster than the (δ, γ)-Skip-Search algorithm which is itself always faster than the Shift-Plus algorithm.

Experiments conduct only on γ-approximation show that an adaptation to this case of the Skip-Search algorithm is faster than an adaptation of the Tuned-Boyer-Moore algorithm.

m	SHIFT-AND	δ-TUNED-BOYER-MOORE	δ-SKIP-SEARCH
8	36.46	26.82	34.64
9	34.46	24.36	31.46
10	33.41	23.61	30.55
20	33.00	22.32	27.54

Table 5: Running times for δ-approximation with $\delta = 9$.

m	SHIFT-PLUS	(δ, γ)-TUNED-BOYER-MOORE	(δ, γ)-SKIP-SEARCH
8	50.73	23.33	31.93
9	50.32	27.78	35.52
10	51.79	33.76	39.45
20	50.26	32.46	36.91

Table 6: Running times for (δ, γ)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 14$.

m	SHIFT-PLUS	(δ, γ)-TUNED-BOYER-MOORE	(δ, γ)-SKIP-SEARCH
8	50.88	23.16	31.99
9	50.86	28.70	36.40
10	51.87	33.74	39.58
20	51.11	32.53	37.38

Table 7: Running times for (δ, γ)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 15$.

m	SHIFT-PLUS	(δ, γ)-TUNED-BOYER-MOORE	(δ, γ)-SKIP-SEARCH
8	50.72	23.33	32.02
9	50.70	27.96	35.65
10	51.94	33.88	40.00
20	51.35	33.20	37.03

Table 8: Running times for (δ, γ)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 16$.

One should notice that the SHIFT-AND and SHIFT-PLUS algorithms need constant time to run whatever the values of the parameters are. In case of very high values for δ and/or γ they have to be considered as the best choice.

m	Shift-Plus	(δ,γ)-Tuned-Boyer-Moore	(δ,γ)-Skip-Search
8	50.67	23.29	32.20
9	50.83	28.38	35.74
10	51.93	34.41	39.91
20	50.18	32.94	37.10

Table 9: Running times for (δ,γ)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 17$.

m	Shift-Plus	(δ,γ)-Tuned-Boyer-Moore	(δ,γ)-Skip-Search
8	51.24	23.57	32.22
9	50.31	28.33	35.73
10	51.83	34.36	40.15
20	49.97	32.77	37.03

Table 10: Running times for (δ,γ)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 18$.

8 Conclusions

Here we presented the Skip-Search, Tuned-Boyer-Moore and Maximal-Shift approximate string matching algorithms that outperform the one presented in [4].

References

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

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.

[3] E. Cambouropoulos, T. Crawford and C.S. Iliopoulos, (1999) Pattern Processing in Melodic Sequences: Challenges, Caveats and Prospects. In Proceedings of the AISB'99 Convention (Artificial Intelligence and Simulation of Behaviour), Edinburgh, U.K., pp. 42-47 (1999).

[4] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 129–144, Perth, WA, Australia, 1999.

[5] C. Charras, T. Lecroq, and J.D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*,

number 1448 in Lecture Notes in Computer Science, pages 55–64, Piscataway, NJ, 1998. Springer-Verlag, Berlin.

[6] T. Crawford, C. S. Iliopoulos, R. Raman, String Matching Techniques for Musical Similarity and Melodic Recognition, *Computing in Musicology*, Vol 11 (1998) 73–100.

[7] V. Fischetti, G. Landau, J.Schmidt and P. Sellers, Identifying periodic occurences of a template with applications to protein structure, *Proc. 3rd Combinatorial Pattern Matching*, Lecture Notes in Computer Science, vol. 644, 1992, pp. 111–120.

[8] A. Hume and D. M. Sunday. Fast string searching. *Software–Practice and Experience*, 21(11):1221–1248, 1991.

[9] S. Karlin, M. Morris, G. Ghandour, and M.-Y. Leung, Efficients algorithms for molecular sequences analysis, *Proc. Natl. Acad. Sci., USA (1988) 85:841–845*

[10] G. Main and R. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, *Journal of Algorithms* **5** (1984), pp. 422–432.

[11] P. McGettrick, MIDIMatch: Musical Pattern Matching in Real Time. MSc Dissertation, York University, U.K. (1997).

[12] A. Milosavljevic and J. Jurka, Discovering simple DNA sequences by the algorithmic significance method, *Comput. Appl. Biosci. (1993) 9:407–411*

[13] P. A. Pevzner and W. Feldman, Gray Code Masks for DNA Sequencing by Hybridization, *Genomics*, 23, 233-235 (1993).

[14] P.Y. Rolland, J.G. Ganascia, Musical Pattern Extraction and Similarity Assessment. In Readings in Music and Artificial Intelligence. E. Miranda. (ed.). Harwood Academic Publishers (forthcoming) (1999).

[15] J. P. Schmidt, All shortest paths in weighted grid graphs and its application to finding all approximate repeats in strings, *SIAM Journal on Computing* 27, 4 (1998), 972-992.

[16] S. S. Skiena and G. Sundaram, Reconstructing strings from substrings, *J. Computational Biol.* 2 (1995) 333–353.

[17] D. M. Sunday, A very fast substring search algorithm, *CACM*, Vol 33, (1990), pp. 132–142.

[18] S. Wu and U. Manber, Fast text searching allowing errors, *CACM*, Vol 35, (1992), pp. 83–91.