Information Processing Letters 111 (2011) 516–520

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

String matching with inversions and translocations in linear average time (most of the time)

Szymon Grabowski [a,*,1], Simone Faro [b], Emanuele Giaquinta [b]

[a] Technical University of Łódź, Computer Engineering Department, Al. Politechniki 11, 90–924 Łódź, Poland
[b] Università di Catania, Dipartimento di Matematica e Informatica, Viale Andrea Doria 6, I-95125 Catania, Italy

ARTICLE INFO

Article history:
Received 1 December 2010
Received in revised form 21 February 2011
Accepted 25 February 2011
Available online 1 March 2011
Communicated by Ł. Kowalik

Keywords:
Approximate string matching
Algorithms
Bioinformatics

ABSTRACT

We present an efficient algorithm for finding all approximate occurrences of a given pattern p of length m in a text t of length n allowing for translocations of equal length adjacent factors and inversions of factors. The algorithm is based on an efficient filtering method and has an $\mathcal{O}(nm\max(\alpha,\beta))$-time complexity in the worst case and $\mathcal{O}(\max(\alpha,\beta,\sigma))$-space complexity, where α and β are respectively the maximum length of the factors involved in any translocation and inversion, and σ is the alphabet size. Moreover we show that our algorithm has an $\mathcal{O}(n)$ average time complexity, whenever $\sigma = \Omega(\log m/\log\log^{1-\varepsilon} m)$, for $\varepsilon > 0$. Experiments show that the proposed algorithm achieves very good results in practical cases.

1. Introduction

Retrieving information and teasing out the meaning of biological sequences are central problems in modern biology. Generally, basic biological information is stored in strings of nucleic acids (DNA, RNA) or amino acids (proteins). With the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application and there is an increasing demand for fast computer methods for analysis and data retrieval.

Approximate string matching is a fundamental problem in text processing and consists in finding approximate matches of a pattern in a string. The closeness of a match is measured in terms of the sum of the costs of the edit operations necessary to convert the string into an exact match. Most classical models, e.g., Levenshtein distance (for a survey, see [1]), assume that changes between strings occur locally. However, evidence shows that large scale changes are possible in chromosomal rearrangement. For example, large pieces of DNA in a chromosomal sequence can be broken and moved from one location to another. This is known as a *chromosomal translocation*. Sometimes a mutation can also flip a stretch of DNA within a chromosome, producing a *chromosomal inversion*.

In particular, a chromosomal inversion is a rearrangement in which a segment of a chromosome is reversed end to end. An inversion occurs when a single chromosome undergoes breakage and rearrangement within itself. Differently, a chromosomal translocation is a chromosome abnormality caused by rearrangement of parts of the same chromosome or between non-homologous chromosomes. Sometimes a chromosomal translocation could join two separated genes, the occurrence of which is common in cancer.

Recently Cantone et al. [2] presented the first solution for the matching problem under a string distance whose edit operations are translocations of equal length adjacent factors and inversions of factors. In particular, they devised an $\mathcal{O}(nm\max(\alpha,\beta))$-time and $\mathcal{O}(m^2)$-space algorithm, where α and β are the maximum length of the factors involved in a translocation and in an inversion, respectively. They showed that under the assumption of equiprobability and independence of characters in the alphabet, on average the algorithm has an $\mathcal{O}(n\log_\sigma m)$-time complexity, where σ is the alphabet size. Moreover they

* Corresponding author.
E-mail address: sgrabow@kis.p.lodz.pl (S. Grabowski).

[1] Supported by the Polish Ministry of Science and Higher Education under the project N N516 441938.

0020-0190/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2011.02.015

```
GFG(p, m, t, n, α, β)
 1.  for c ∈ Σ do G[c] ← 0
 2.  for s ← 0 to m − 1 do
 3.    G[p[s]] ← G[p[s]] + 1
 4.    G[t[s]] ← G[t[s]] − 1
 5.  δ ← 0
 6.  for c ∈ Σ do δ ← δ + abs(G[c])
 7.  for s ← 0 to n − m do
 8.    if δ = 0 then
 9.      VERIFY(p, m, t, s, α, β)
10.    a ← t[s]
11.    b ← t[s + m]
12.    δ ← δ − abs(G[a]) − abs(G[b])
13.    G[a] ← G[a] + 1
14.    G[b] ← G[b] − 1
15.    δ ← δ + abs(G[a]) + abs(G[b])
16.  if δ = 0 then
17.    VERIFY(p, m, t, n − m, α, β)
```

```
VERIFY(p, m, t, s, α, β)
 1.  γ = min(α, β)
 2.  for i ← 0 to m − 1 do
 3.    for j ← max(0, i − γ) to min(m − 1, i + γ) do
 4.      F[i, j] ← I[i, m − j − 1] ← 0
 5.      if (p[i] = t[s + j]) then
 6.        if (i = 0 or j = 0) then F[i, j] ← 1
 7.        else F[i, j] ← F[i − 1, j − 1] + 1
 8.      if (p[i] = t[s + m − j − 1]) then
 9.        if (i = 0 or j = 0) then I[i, m − j − 1] ← 1
10.        else I[i, m − j − 1] ← I[i − 1, m − j] + 1
11.    if (p[i] = t[s + i] and (i = 0 or S[i − 1] = 1))
12.    then S[i] ← 1 else S[i] ← 0
13.    for k ← 1 to min(α, ⌊(i+1)/2⌋) do
14.      if (F[i, i − k] ⩾ k and F[i − k, i] ⩾ k) then
15.        if (i < 2k or S[i − 2k] = 1) then S[i] ← 1
16.    for k ← 2 to min(β, i + 1) do
17.      if (I[i, i − k + 1] ⩾ k) then
18.        if (i < k or S[i − k] = 1) then S[i] ← 1
19.  if (S[m − 1] = 1) then Output(s)
```

Fig. 1. (on the left) The GFG algorithm for the approximate string matching problem with inversions and translocations and (on the right) the verification procedure.

also presented a bit-parallel implementation of their algorithm, working in $\mathcal{O}(n\max(\alpha,\beta))$ time and $\mathcal{O}(\sigma+m)$ space, if the pattern length is comparable with the computer word size.

In this paper we present a new algorithm for the same problem based on an efficient permutation filtering method and on a dynamic programming approach for testing candidate positions. In particular our algorithm achieves an $\mathcal{O}(nm\max(\alpha,\beta))$-worst case time complexity, as the M-SAMPLING algorithm, and requires only $\mathcal{O}(\max(\alpha,\beta))$ space. More interestingly, our algorithm is shown to achieve $\mathcal{O}(n)$ average-case time complexity, for $\sigma=\Omega(\log m/\log\log^{1-\varepsilon} m)$, $\varepsilon>0$.

2. Basic notions and definitions

Let $p[0\ldots m-1]$ be a string of length $len(p)=m\geqslant 0$, over an integer alphabet Σ of size σ. We denote by $p[i]$ the $(i+1)$th character of p. Likewise, the substring (also called *factor*) of p contained between the $(i+1)$th and the $(j+1)$th characters of p is indicated with $p[i\ldots j]$, for $i\leqslant j$. An k-substring (or k-factor) is a substring of length k. In addition, we write pp' to denote the concatenation of p and p', and p^r for the reverse of the string p.

A *distance* $d:\Sigma^*\times\Sigma^*\to\mathbb{R}$ is a function which associates to any pair of strings X and Y the minimal cost of any finite sequence of edit operations which transforms X into Y, if such a sequence exists, ∞ otherwise.

Definition 1. Given two strings X and Y, the *mutation distance* $md(X,Y)$ is based on the following edit operations:

(1) **Inversion**: a factor Z is transformed into Z^r.
(2) **Translocation**: a factor of the form ZW is transformed into WZ, provided that $len(Z)=len(W)>0$.

Both operations are assigned unit cost.

We indicate with α and β the maximum length of factors involved translocations and inversions, respectively. By definition, $\alpha\leqslant\lfloor len(X)/2\rfloor$ and $\beta\leqslant len(X)$. When $md(X,Y)<\infty$, we say that X and Y have an md-match. Additionally, if X has an md-match with a suffix of Y, we write $X\sqsupseteq_{md} Y$.

Definition 2. The string matching with inversions and translocations (SMIT) problem: Given text $t[0\ldots n-1]$ and pattern $p[0\ldots m-1]$, $n\geqslant m$, return all text positions s such that $p\sqsupseteq_{md} t[0\ldots s+m-1]$.

3. Proposed algorithm

In this section we present a new efficient algorithm for the approximate string matching problem allowing for inversions of factors and translocations of equal length adjacent factors. In the following we assume that p and t are strings of length m and n respectively, over a common alphabet $\Sigma=\{c_0,\ldots,c_{\sigma-1}\}$, where $\sigma=O(n)$. (The case of even larger alphabets is rather theoretical and can be handled with standard solutions, e.g., using a minimal perfect hash function.)

The new algorithm, named GFG algorithm, searches for all occurrences of p in t by making use of an efficient filter method. This technique, usually called as the *counting filter*, is known in the literature [3–5] and has been used for k-mismatches and k-differences. The idea behind the filter is straightforward and is based upon the observation that (in our problem) if the pattern p has an occurrence (possibly involving inversions and translocations) starting at position s of the text then $t[s\ldots s+m-1]$ is a permutation of the pattern.

In short, the GFG algorithm identifies the set $\Gamma_{p,t}$ of all candidate positions s in the text such that the substring $t[s\ldots s+m-1]$ is a permutation of the characters in p and, for each $s\in\Gamma_{p,t}$, executes a verification procedure in order to check the approximate occurrence.

As the counting filter technique is quite well known, we refer the reader to the pseudocode of the GFG algorithm in Fig. 1 (on the left) and the verification procedure (on the right), providing only brief comments below.

For each position $0 \leqslant s \leqslant n-m$, we define a function $G_s : \Sigma \to N$, as $G_s(c) = occ_p(c) - occ_{t(s,m)}(c)$ for $c \in \Sigma$, and where we set $t(s,m) = t[s \ldots s+m-1]$.

We also define, for each position s, the distance value δ_s as follows

$$\delta_s = \delta(p, t_s) = \sum_{c\in\Sigma} abs\big(occ_p(c) - occ_{t(s,m)}(c)\big) = \sum_{c\in\Sigma} abs\big(G_s(c)\big)$$

Then it is easy to see that the set $\Gamma_{p,t}$ of all candidate positions in the text can be defined as $\Gamma_{p,t} = \{s \mid 0 \leqslant s \leqslant n-m \text{ and } \delta_s = 0\}$.

Observe that values δ_{s+1} and δ_s can differ only in the number of occurrences of characters $t[s]$ and $t[s+m]$. Thanks to this property, $G_{s+1}(c)$ can be computed in constant time from $G_s(c)$ and similarly δ_{s+1} can be computed from δ_s in constant time (lines 12–15 in Fig. 1, left).

Note that the main loop of GFG has only one conditional and the integer *abs* function is translated by modern compilers (including `GNU C Compiler`) into branchless code.

The verification procedure is based on dynamic programming. The algorithm uses two matrices, F and I, both of size m^2, in order to compute occurrences of factors and inverted factors of p, respectively, in the substring $t[s \ldots s+m-1]$. More formally we define

$$F[i,j] = \max\big\{k \mid p[i-k+1\ldots i] = t[s+j-k+1\ldots s+j]\big\}$$

and

$$I[i,j] = \max\big\{k \mid p[i-k+1\ldots i] = \big(t[s+j\ldots s+j+k-1]\big)^r\big\}$$

for $0 \leqslant i < m$ and $\max(0, i-\gamma) \leqslant j \leqslant \min(m-1, i+\gamma)$, where $\gamma = \min(\alpha, \beta)$. Moreover a vector S, of size m, is maintained in order to compute the *md*-matches of all prefixes of the pattern in $t[s \ldots s+m-1]$. More formally, for $0 \leqslant i < m$, we have $S[i] = 1$ if $p_i \sqsupseteq_{md} t[s \ldots s+i]$ and $S[i] = 0$ otherwise.

The following recursive relations are used for computing F and I.

$$F[i,j] = \begin{cases} 0 & \text{if } p[i] \neq t[s+j] \\ F[i-1, j-1]+1 & \text{if } i > 0,\ j > \max(0, i-\alpha) \\ & \text{and } p[i] = t[s+j] \\ 1 & \text{otherwise} \end{cases}$$

$$I[i,j] = \begin{cases} 0 & \text{if } p[i] \neq t[s+j] \\ I[i-1, j+1]+1 & \text{if } i > 0, \\ & j < \min(m-1, i+\beta) \\ & \text{and } p[i] = t[s+j] \\ 1 & \text{otherwise} \end{cases}$$

Finally the vector S is computed, for increasing $i = 0 \ldots m-1$ according to the following (recursive) formula. The value of $S[i]$ is set to 1 iff one of the following conditions holds:

- $p[i] = t[s+i]$ and ($i = 0$ or $S[i-1] = 1$);
- $F[i, i-k] \geqslant k$, $F[i-k, i] \geqslant k$ and ($i < 2k$ or $S[i-2k] = 1$), for $1 \leqslant k \leqslant \min(\alpha, \lfloor \frac{i+1}{2} \rfloor)$;
- $I[i, i-k+1] \geqslant k$ and ($i < k$ or $S[i-k] = 1$), for $1 \leqslant k \leqslant \min(\beta, i+1)$.

Then p has an *md*-match starting at position s of the text if $S[m-1] = 1$ at the end of the verification procedure with parameter p, t and s.

Observe that the computation of the entry of position i in S only requires the last β entries of the $(i-1)$th row of I and the last α entries of both the $(i-1)$th row of F and $(i-1)$th column of F. Similarly only the last $\max(2\alpha, \beta)$ entries of the vector S are needed for computing the value $S[i]$. Moreover, both for I and F, the computation of the ith row (column) needs only the values in the $(i-1)$th row (column) of the matrix.

It is thus straightforward to reduce the space requirements of the verification phase to $\mathcal{O}(\max(\alpha, \beta))$. This is done by maintaining, for each iteration, only two rows of I and only two rows and two columns of F, each of size $\max(\alpha, \beta)$.

The verification time and space costs are thus $\mathcal{O}(m \max(\alpha, \beta))$ and $\mathcal{O}(\max(\alpha, \beta))$, respectively, leading to overall $\mathcal{O}(nm \max(\alpha, \beta))$ worst case time complexity and $\mathcal{O}(\max(\alpha, \beta, \sigma))$ space complexity for the GFG algorithm.

Finally, note that the arrays F and I store results of the longest common extension (LCE) queries and for this problem efficient theoretical results are known. More precisely, one can build a suffix tree over $p\#t$, where # is a symbol not occurring in Σ, and preprocess it for lowest common ancestor (LCA) queries. This can be done once for the whole text in $\mathcal{O}(n+m)$ time, using $\mathcal{O}(n)$ space, and the queries are handled in constant time [6]. The space use can be decreased to $\mathcal{O}(m)$ using a standard technique ($p\#$ concatenated with overlapping windows of t of width $2m-1$) without compromising the overall build and query answering time complexities. Still, in this case the calculations of vector S in our procedure remain the (theoretical) bottleneck, and the overall complexities are unchanged (the minor difference in space, between $\mathcal{O}(\max(\alpha, \beta))$ of our solution and $\mathcal{O}(m)$ of the suffix tree based one, is practically irrelevant, since usually the whole text of length $\mathcal{O}(n)$ is stored in the main memory as well). Our solution based on the arrays F and I is conceptually simpler and does not suffer from large hidden constants associated with suffix trees.

4. Average-case time analysis

Next, we evaluate the average time complexity of the GFG algorithm. In our analysis we assume the uniform distribution and independence of characters. We first assume that $m = \omega(\sigma^{\mathcal{O}(1)})$, then we prove the simple case when $m \leqslant \sigma$.

Our verification procedure takes $\mathcal{O}(m^2)$ (worst-case) time per location. To obtain linear average time, we must thus bound the probability of having permuted subse-

Table 1
The performance of M-Sampling [2] (MS), GFG using M-Sampling for verification (GFG1) and GFG as shown in Fig. 1 (GFG2).

Random text with $\sigma = 4$			
m	MS	GFG1	GFG2
8	254.78	48.53	73.73
16	350.25	50.05	103.09
32	441.05	44.20	102.04
64	528.35	43.83	140.18
128	645.36	43.20	208.05
256	868.13	41.84	273.47
512	1273.13	44.71	349.57

Random text with $\sigma = 8$			
m	MS	GFG1	GFG2
8	155.39	29.57	29.78
16	193.91	29.21	28.98
32	241.54	29.20	28.72
64	309.26	29.33	28.75
128	377.17	29.68	29.16
256	525.96	30.75	30.89
512	770.45	34.14	37.73

Random text with $\sigma = 16$			
m	MS	GFG1	GFG2
8	115.27	28.45	28.55
16	137.27	28.48	28.54
32	161.25	28.51	28.57
64	211.75	28.65	28.66
128	273.53	28.94	29.01
256	371.65	29.86	30.34
512	536.40	32.85	35.79

Random text with $\sigma = 32$			
m	MS	GFG1	GFG2
8	93.80	28.18	28.52
16	110.64	28.20	28.53
32	128.80	28.25	28.55
64	169.25	28.42	28.61
128	197.24	28.65	28.93
256	259.77	29.45	30.23
512	398.20	32.07	35.11

Escherichia coli			
m	MS	GFG1	GFG2
8	593.49	117.79	184.48
16	781.76	108.53	208.50
32	976.79	99.88	222.19
64	1188.58	94.64	267.01
128	1484.03	84.16	252.17
256	2005.00	80.40	257.70
512	2929.90	83.36	299.49

Saccharomyces cerevisiae			
m	MS	GFG1	GFG2
8	163.25	41.38	41.45
16	192.64	41.39	41.45
32	224.27	41.44	41.48
64	297.01	41.56	41.60
128	376.27	41.88	41.91
256	506.88	42.79	43.25
512	738.19	45.72	48.65

quences of length m with $\mathcal{O}(1/m^2)$. We will find conditions upon which this happens.[2]

Suppose $m = \omega(\sigma^{\mathcal{O}(1)})$, we define $k = m/\sigma$ and, without loss of generality, we assume that σ divides m. For each text position s, with $0 \leqslant s \leqslant n - m$, the probability that the m-substring of the text, beginning at position s, is a permutation of the pattern p is exactly

$$\Pr\{s \in \Gamma_{p,t}\} = \frac{\binom{m}{occ(c_0)}\binom{m-occ(c_0)}{occ(c_1)}\binom{m-occ(c_0)-occ(c_1)}{occ(c_2)}\cdots\binom{occ(c_{\sigma-1})}{occ(c_{\sigma-1})}}{\sigma^m} \tag{1}$$

Now, it is easy to notice that the probability given in (1) is maximized when $occ(c_i) = k$ for all i. We can thus write:

$$\Pr\{s \in \Gamma_{p,t}\} \leqslant \frac{\binom{m}{k}\binom{m-k}{k}\binom{m-2k}{k}\cdots\binom{k}{k}}{\sigma^m} = \frac{m!}{(k!)^\sigma \sigma^m}$$

We make use of Stirling's approximation for $m!$ and $k!$ (recall that $k = m/\sigma$):

$$\frac{m!}{(k!)^\sigma \sigma^m} = \Theta\left(\frac{\sqrt{2\pi m}(m/e)^m}{(\sqrt{2\pi(m/\sigma)}(m/(e\sigma))^{m/\sigma})^\sigma \sigma^m}\right) = \Theta\left(\frac{\sqrt{2\pi m}}{(\sqrt{2\pi(m/\sigma)})^\sigma}\right)$$

Let us upper-bound $\sqrt{2\pi}/(\sqrt{2\pi})^\sigma$ with 1 and remove it. We have:

$$\Theta\left(\frac{\sqrt{m}}{(\sqrt{m/\sigma})^\sigma}\right) = \Theta\left(\frac{\sigma^{\sigma/2}}{m^{(\sigma-1)/2}}\right)$$

Let us assume $m \geqslant \sigma^4$ (we recall that $m = \omega(\sigma^{\mathcal{O}(1)})$). Then $\sigma^{\sigma/2}/m^{(\sigma-1)/2}$ is less than or equal to $1/\sigma^{1.5\sigma-2}$. Note that if we take a larger lower bound on m, e.g., σ^8, then our upper bound gets even smaller, namely $1/\sigma^{3.5\sigma-4}$ in this example. All in all, we have

$$\Pr\{s \in \Gamma_{p,t}\} = \mathcal{O}\big(1/\sigma^{\mathcal{O}(\sigma)}\big) = \mathcal{O}\big(1/m^2\big)$$

for any $\sigma = \Omega(\log m/\log\log^{1-\varepsilon} m)$, where $\varepsilon > 0$.

Suppose now that $m \leqslant \sigma$.[3] Then the probability that the m-substring of the text, beginning at position s, is a permutation of the pattern p is

$$\Pr\{s \in \Gamma_{p,t}\} \leqslant \frac{m!}{\sigma^m} \leqslant \frac{m!}{m^m} < \sqrt{2\pi}\frac{m^{m+1}}{e^m m^m} = \sqrt{2\pi}\frac{m}{e^m} = \mathcal{O}\big(1/m^2\big)$$

where we made use again of Stirling's approximation for $m!$.

Thus the overall average time complexity of the GFG algorithm, assuming $\sigma = \Omega(\log m/\log\log^{1-\varepsilon} m)$, is given by the following relation:

$$\begin{aligned} T(n,m,\sigma) &= \mathcal{O}(\sigma + m) + \sum_{s=0}^{n-m} \Pr\{s \in \Gamma_{p,t}\} \cdot \mathcal{O}\big(m^2\big) \\ &= \mathcal{O}(\sigma + m) + (n - m + 1) \cdot \mathcal{O}\big(1/m^2\big) \cdot \mathcal{O}\big(m^2\big) \\ &= \mathcal{O}(n) \end{aligned}$$

5. Experimental results

We evaluate the performance of the following algorithms: M-Sampling [2] (MS), GFG using M-Sampling for verification (GFG1) and GFG as shown in Fig. 1 (GFG2) (see Table 1). All algorithms have been implemented in C and compiled with the `GNU C Compiler 4.2`, using the options `-O2 -fno-guess-branch-probability`. All tests have been performed on a 2 GHz Intel Core

[2] The paper [5] contains an analysis of the counting filter, in the k-differences problem. Unfortunately, the analysis seems to be flawed, which was admitted in discussion by the second author of the cited paper (G. Navarro).

[3] Note that for the more general case of $m = \sigma^{\mathcal{O}(1)}$ there exists already an average-case linear algorithm [2], so this part of the analysis is only to find properties of the currently presented algorithm.

Table 2
The mean, over the 200 runs, of the number of pattern's permutations found per text position.

Random text ($\sigma = 4$)	
m	# candidate
8	0.013621
16	0.006399
32	0.001837
64	0.000720
128	0.000285
256	0.000093
512	0.000029

Random text ($\sigma = 8$)	
m	# candidate
8	0.000410
16	0.000037
32	0.000004
64	0.000001
128	0.000001
256	0.000001
512	0.000001

Random text ($\sigma = 16$)	
m	# candidate
8	0.000004
16	0.000001
32	0.000001
64	0.000001
128	0.000001
256	0.000001
512	0.000001

2 Duo and running times have been measured with a hardware cycle counter, available on modern CPUs. We used the following input files: (i) four random texts of 2,000,000 characters with a uniform distribution over alphabets of dimension σ, with $\sigma \in \{4, 8, 16, 32\}$ respectively; (ii) a protein sequence of 2,900,352 characters from the *Saccharomyces cerevisiae* genome (with $\sigma = 20$)[4]; (iii) a genome sequence of 4,638,690 base pairs of *Escherichia coli* ($\sigma = 4$).[5]

For each input file, we have generated seven sets of 200 patterns of fixed length m randomly extracted from the text, for $m \in \{8, 16, 32, 64, 128, 256, 512\}$. For each set of patterns we reported the mean time over 200 runs, expressed in ms.

The experimental results show that the filtering strategy is quite effective and allows to dramatically speed up, by a factor of at most 30, the computation of the *md*-matches. For very small alphabets the GFG1 algorithm, based on M-Sampling, is faster than GFG2, based on the dynamic programming verification, while in the other cases the two algorithms have almost the same speed.

In Table 2 we report the mean, over the 200 runs, of the number of pattern's permutations found per text position.

Observe that, while for small alphabets the number is non-negligible also for long patterns, for large enough alphabets it is always insignificant.

Acknowledgements

We thank an anonymous referee for constructive comments on the manuscript.

References

[1] G. Navarro, A guided tour to approximate string matching, ACM Comput. Surv. 33 (1) (2001) 31–88.
[2] D. Cantone, S. Faro, E. Giaquinta, Approximate string matching allowing for inversions and translocations, in: J. Holub, J. Žd'árek (Eds.), Proceedings of the Prague Stringology Conference, Czech Technical University, Prague, Czech Republic, 2010, pp. 37–51.
[3] R. Grossi, F. Luccio, Simple and efficient string matching with k mismatches, Inform. Process. Lett. 33 (3) (1989) 113–120.
[4] P. Jokinen, J. Tarhio, E. Ukkonen, A comparison of approximate string matching algorithms, Softw. Pract. Exp. 26 (12) (1996) 1439–1458.
[5] R.A. Baeza-Yates, G. Navarro, New and faster filters for multiple approximate string matching, Random Structure Algorithms 20 (1) (2002) 23–49.
[6] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997.

[4] http://data-compression.info/Corpora/ProteinCorpus/.
[5] http://corpus.canterbury.ac.nz/.