String Matching with Inversions and Translocations in Linear Average Time (Most of the Time)

Szymon Grabowski[†], Simone Faro[‡], and Emanuele Giaquinta[‡]

[†] Computer Engineering Department, Technical University of Łódź,
Al. Politechniki 11, 90-924 Łódź, Poland
sgrabow@kis.p.lodz.pl

[‡] Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{faro | giaquinta}@dmi.unict.it

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 $\mathcal{O}(\max(\alpha,\beta))$-space complexity, where α and β are respectively the maximum length of the factors involved in any translocation and inversion. Moreover we show that under the assumptions of equiprobability and independence of characters our algorithm has a $\mathcal{O}(n)$ average time complexity, whenever $\sigma = \Omega(\log m / \log\log^{1-\varepsilon} m)$, where $\varepsilon > 0$ and σ is the dimension of the alphabet. Experiments show that the new 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). Aligning sequences helps in revealing their shared characteristics, while matching sequences can infer useful information from them. 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 or Damerau edit distance (for a survey see [5]) assume that changes between strings occur locally. However, evidence shows that large scale changes are possible in chromosomal rearrangment. For example, large pieces of DNA in

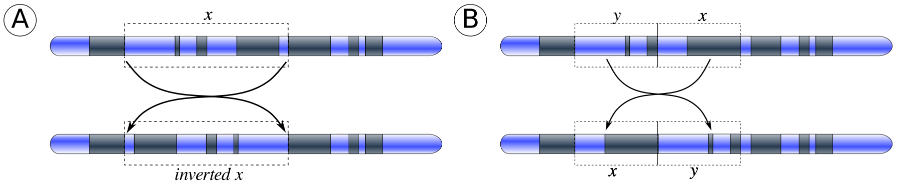

Fig. 1. An example of chromosomal inversion.

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. Fig. 1(A) shows an example of chromosomal inversion.

Differently a chromosomal translocation is a chromosome abnormality caused by rearrangement of parts of the same chromosome or between nonhomologous chromosomes. Sometimes a chromosomal translocation could join two separated genes, the occurrence of which is common in cancer. Fig. 1(B) shows an example of chromosomal translocation.

Both inversions and translocations do not involve a loss of genetic information, but simply rearrange the linear gene sequence.

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 a $\mathcal{O}(nm \max(\alpha, \beta))$-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 a $\mathcal{O}(n \log_\sigma m)$-time complexity. Moreover they also presented a bit-parallel implementation of their algorithm, which has $\mathcal{O}(n \max(\alpha, \beta))$-time and $\mathcal{O}(\sigma+m)$-space complexity, 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 a $\mathcal{O}(nm \max(\alpha, \beta))$-worst case time complexity, as the M-Sampling algorithm, and requires only $\mathcal{O}(\max(\alpha, \beta))$ space. Moreover we show that under the assumption of equiprobability and independence of characters in the alphabet, on the average our filter based algorithm achieves a $\mathcal{O}(n)$-time complexity, when $\sigma = \Omega(\log m / \log\log^{1-\varepsilon} m)$, where $\varepsilon > 0$ and σ is the dimension of the alphabet.

A slightly shorter version of this manuscript was submitted to *Information Processing Letters*.

2 Basic notions and definitions

Let p be a string of length $m \geq 0$, over an integer alphabet Σ of size σ. We represent it as a finite array $p[0 \ldots m-1]$ of characters from Σ and write $len(p) = m$. In particular, for $m = 0$ we obtain the empty string ε. We denote by $p[i]$ the $(i+1)$th character of p, for $0 \leq i < m$. 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 $0 \leq i \leq j < m$. An m-substring (or m-factor) is a substring of length m. We also put $p_i =_{\mathrm{Def}} p[0 \ldots i]$, for $0 \leq i < m$. In addition, we write pp' to denote the concatenation of p and p', and p^{r} for the reverse of the string p, i.e., $p^{\mathsf{r}} =_{\mathrm{Def}} p[m-1]p[m-2] \ldots p[0]$. Given a string p and a character $c \in \Sigma$, we define $occ_p(c)$ as the number of times the character c occurs in p (observe that $0 \leq occ_p(c) \leq len(p)$).

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

Both operations are assigned unit cost. □

We indicate with α and β the maximum length of factors involved translocations and inversions, respectively. By definition, $\alpha \leq \lfloor len(X)/2 \rfloor$ and $\beta \leq len(X)$. 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 $X \sqsupseteq_{md} Y$.

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, 4, 1] 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 approximate occurrence (possibly involving inversions and translocations) starting at position s of the text then the m-substring of the text $t[s \ldots s+m-1]$ is a permutation of the pattern.

Then 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.

Before entering into details we need to introduce some additional notations. Given two strings w and z, we define a distance function $\delta(w, z)$ as

$$\delta(w,z) = \sum_{c \in \Sigma} abs\big(occ_w(c) - occ_z(c)\big).$$

Obviously, if $len(w) = len(z)$, then $\delta(w,z) = 0$ iff w is a permutation of z.

For each position s in the text, with $0 \leq s \leq n-m$, we define a function $G_s : \Sigma \rightarrow N$, as

$$G_s(c) = occ_p(c) - occ_{t(s,m)}(c)$$

for $c \in \Sigma$, and where we set $t(s,m) = t[s \ldots s+m-1]$.

Finally we 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 the set $\Gamma_{p,t}$ of all candidate positions in the text can be defined as

$$\Gamma_{p,t} = \{s \mid 0 \leq s \leq n-m \text{ and } \delta_s = 0\}.$$

Observe that values δ_{s+1} and δ_s can differ only in the number of occurrences of characters $t[s]$ and $t[s+m]$. Specifically we have $occ_{t(s+1,m)}(t[s]) \geq occ_{t(s,m)}(t[s]) - 1$ and $occ_{t(s+1,m)}(t[s+m]) \leq occ_{t(s,m)}(t[s+m]) + 1$. Moreover, if $t[s] = t[s+m]$, the two functions $occ_{t(s+1,m)}$ and $occ_{t(s,m)}$ do not differ for any value.

Therefore, for each character $c \in \Sigma$, the value of $G_{s+1}(c)$ can be computed in constant time from $G_s(c)$ by using the following relation

$$G_{s+1}(c) = \begin{cases} G_s(c) - 1 & \text{if } c = t[s+m] \neq t[s] \\ G_s(c) + 1 & \text{if } c = t[s] \neq t[s+m] \\ G_s(c) & \text{otherwise} \end{cases}$$

which gives the following relation for computing δ_{s+1} from δ_s in constant time

$$\begin{aligned} \delta_{s+1} = \delta_s - abs\big(G_s(t[s])\big) - abs\big(G_s(t[s+m])\big) + \\ + abs\big(G_{s+1}(t[s])\big) + abs\big(G_{s+1}(t[s+m])\big). \end{aligned}$$

Fig.2 shows the pseudocode of the GFG algorithm (on the left) and the verification procedure (on the right). 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

```
GFG $(p, m, t, n, \alpha, \beta)$
for $c \in \Sigma$ do $G[c] \leftarrow 0$
for $s \leftarrow 0$ to $m - 1$ do
    $G[p[s]] \leftarrow G[p[s]] + 1$
    $G[t[s]] \leftarrow G[t[s]] - 1$
$\delta \leftarrow 0$
for $c \in \Sigma$ do $\delta \leftarrow \delta + abs(G[c])$
for $s \leftarrow 0$ to $n - m$ do
    if $\delta = 0$ then
        Verify$(p, m, t, s, \alpha, \beta)$
    $a \leftarrow t[s]$
    $b \leftarrow t[s + m]$
    $\delta \leftarrow \delta - abs(G[a]) - abs(G[b])$
    $G[a] \leftarrow G[a] + 1$
    $G[b] \leftarrow G[b] - 1$
    $\delta \leftarrow \delta + abs(G[a]) + abs(G[b])$
if $\delta = 0$ then
    Verify$(p, m, t, n - m, \alpha, \beta)$
```

```
Verify$(p, m, t, s, \alpha, \beta)$
$\gamma = \min(\alpha, \beta)$
for $i \leftarrow 0$ to $m - 1$ do
    for $j \leftarrow \max(0, i - \gamma)$ to $\min(m - 1, i + \gamma)$ do
        $F[i, j] \leftarrow I[i, m - j - 1] \leftarrow 0$
        if $(p[i] = t[s + j])$ then
            if $(i = 0$ or $j = 0)$ then $F[i, j] \leftarrow 1$
            else $F[i, j] \leftarrow F[i - 1, j - 1] + 1$
        if $(p[i] = t[s + m - j - 1])$ then
            if $(i = 0$ or $j = 0)$ then $I[i, m - j - 1] \leftarrow 1$
            else $I[i, m - j - 1] \leftarrow I[i - 1, m - j] + 1$
    if $(p[i] = t[s + i]$ and $(i = 0$ or $S[i - 1] = 1))$
    then $S[i] \leftarrow 1$ else $S[i] \leftarrow 0$
    for $k \leftarrow 1$ to $\min(\alpha, \lfloor \frac{i+1}{2} \rfloor)$ do
        if $(F[i, i - k] \geq k$ and $F[i - k, i] \geq k)$ then
            if $(i < 2k$ or $S[i - 2k] = 1)$ then $S[i] \leftarrow 1$
    for $k \leftarrow 2$ to $\min(\beta, i + 1)$ do
        if $(I[i, i - k + 1] \geq k)$ then
            if $(i < k$ or $S[i - k] = 1)$ then $S[i] \leftarrow 1$
if $(S[m - 1] = 1)$ then Output$(s)$
```

Fig. 2. (on the left) The GFG algorithm for the approximate string matching problem with inversions and translocations and (on the right) the verification procedure.

factors and inverted factors of p, respectively, in the substring $t[s \ldots s + m - 1]$. More formally we define

$$\begin{aligned} F[i, j] &= \max\{k \mid p[i - k + 1 \ldots i] = t[s + j - k + 1 \ldots s + j]\}, \text{ and} \\ I[i, j] &= \max\{k \mid p[i - k + 1 \ldots i] = (t[s + j \ldots s + j + k - 1])^r\} \end{aligned}$$

for $0 \leq i < m$ and $\max(0, i - \gamma) \leq j \leq \min(m - 1, i + \gamma)$, 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 $0 \leq i < m$, 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 $i = 1 \ldots m - 1$ ($S[i]$ is set to 0) 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 $S[i - 1] = 1)$;
- $F[i, i - k] \geq k$, $F[i - k, i] \geq k$ and $(i < 2k$ or $S[i - 2k] = 1)$, for $1 \leq k \leq \min(\alpha, \lfloor \frac{i+1}{2} \rfloor)$;
- $I[i, i - k + 1] \geq k$ and $(i < k$ or $S[i - k] = 1)$, for $1 \leq k \leq \min(\beta, i + 1)$.

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 for computing the entry of position i in S only the last β entries of the $(i-1)$th row of I are needed, while only the last α entries of the $(i-1)$th row of F and of the $(i-1)$th column of F are needed. 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.

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 more simple case when $m \leq \sigma$.

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

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 $0 \leq s \leq n-m$, the probability that the m-substring of the text, beginning at position s, is a permutation of the pattern p is exactly

$$\Pr\{s \in \mathit{\Gamma}_{p,t}\} = \frac{\binom{m}{occ(c_0)}\binom{m-occ(c_0)}{occ(c_1)}\binom{m-occ(c_0)-occ(c_1)}{occ(c_2)}\cdots\binom{occ(c_{\sigma-1})}{occ(c_{\sigma-1})}}{\sigma^m}. \qquad (1)$$

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:

$$\Pr\{s \in \mathit{\Gamma}_{p,t}\} \leq \frac{\binom{m}{k}\binom{m-k}{k}\binom{m-2k}{k}\cdots\binom{k}{k}}{\sigma^m} = \frac{m!}{(k!)^\sigma \sigma^m}.$$

We make use of Stirling's approximation for both $m!$ and $k!$, recalling that $k = m/\sigma$:

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

[1] The paper [1] 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).

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

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

Let us assume $m \geq \sigma^4$ (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

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

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

Suppose now that $m \leq \sigma$.[2] Then the probability that the m-substring of the text, beginning at position s, is a permutation of the pattern p is

$$\Pr\{s \in \Gamma_{p,t}\} \leq \frac{m!}{\sigma^m} \leq \frac{m!}{m^m}$$

If we make use again of Stirling's approximation for $m!$, we obtain

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

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:

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

5 Experimental results

In this section we evaluate the performance of the following algorithms:

- The M-Sampling algorithm [2] (`MS`)
- The GFG algorithm using the M-Sampling algorithm for verification (`GFG1`)
- The GFG algorithm as shown in Fig.2 (`GFG2`)

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 Duo and running times have been measured with a hardware cycle counter, available on modern CPUs. We used the following input files:

[2] 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.

(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 $\sigma = 20$),[3]
(iii) a genome sequence of $4,638,690$ base pairs of *Escherichia coli* (with $\sigma = 4$).[4]

For each input file, we have generated seven sets of 200 patterns of fixed length m randomly extracted from the text (with at least one occurrence in the text), for m ranging over the values 8, 16, 32, 64, 128, 256, 512. For each set of patterns we reported the mean time over 200 runs, expressed in milliseconds.

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

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 of a given pattern. It is worth observing that for very small alphabets the GFG1 algorithm, based on M-Sampling, is faster than the GFG2 algorithm, based on the dynamic programming verification, while in the other cases the two algorithms have almost the same speed. In the following tables we report 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

Average number of candidate positions for each text character on random texts with $\sigma = 4$ (on the left) $\sigma = 8$ (in the center) $\sigma = 16$ (on the right)

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

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

6 Acknowledgement

The work was partially supported (the first author) by the Polish Ministry of Science and Higher Education under the project N N516 441938.

References

1. R. A. Baeza-Yates and G. Navarro. New and faster filters for multiple approximate string matching. *Random Struct. Algorithms*, 20(1):23–49, 2002.
2. D. Cantone, S. Faro, and E. Giaquinta. Approximate string matching allowing for inversions and translocations. In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference 2010*, pages 37–51, Czech Technical University in Prague, Czech Republic, 2010.
3. R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Inf. Process. Lett.*, 33(3):113–120, 1989.
4. P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Softw. Pract. Exp.*, 26(12):1439–1458, 1996.
5. G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.