Proceedings of the 2012 IEEE 12th International Conference on Bioinformatics & Bioengineering (BIBE), Larnaca, Cyprus, 11-13 November 2012

Fast Searching in Biological Sequences Using Multiple Hash Functions

Simone Faro
Dip. di Matematica e Informatica, Università di Catania
Viale A.Doria n.6, 95125 Catania, Italy
Email: faro@dmi.unict.it

Thierry Lecroq
Université de Rouen, LITIS EA 4108
76821 Mont-Saint-Aignan Cedex, France
Email: thierry.lecroq@univ-rouen.fr

***Abstract*—With the availability of large amounts of DNA data, exact matching of nucleotide sequences has become an important application in modern computational biology and in metagenomics. In this paper we present an efficient method based on multiple hashing functions which improves the performance of existing string matching algorithms when used for searching DNA sequences. From our experimental results it turns out that the new proposed technique leads to algorithms which are up to 8 times faster than the best algorithm known for matching multiple patterns. It turns out also that the gain in performances is larger when searching for larger sets. Thus, considering the fact that the number of reads produced by next generation sequencing equipments is ever growing, the new technique serves a good basis for massive multiple long pattern search applications.**

***Index Terms*—string matching, DNA searching, text processing, biological sequences, hashing algorithms.**

I. Introduction

In molecular biology, nucleotide sequences are the fundamental information for each species and a comparison between such sequences is an interesting and basic problem. Generally 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 [12]. There are various kinds of comparison tools which provide aligning and approximate matching (see for instance [15], [12]), however most of them are based on exact matching in order to speed up the process. Moreover exact string matching is widely used in computational biology for a variety of other tasks. Thus the need for fast matching algorithms on DNA sequences.

In this article we consider the problem of searching for all exact occurrences of a set of r patterns $P = \{p_0, p_1, \ldots, p_{r-1}\}$ in a text t, of length n. We focus on the case where the text t and the patterns p_i are DNA sequences over a finite alphabet $\Sigma = \{\mathsf{a}, \mathsf{c}, \mathsf{g}, \mathsf{t}\}$ of constant size $\sigma = 4$. We are interested here in the problem where the set of patterns is given first and can then be searched in various texts, thus a preprocessing phase is allowed on the patterns. This problem is referred in literature as the *multiple string matching problem*.

In particular we present an improvement of a filtering method based on hashing and q-grams which provides good performances in practical cases for matching DNA sequences. This kind of solutions uses a hashing function in order to locate candidate occurrences along the text and, whenever a candidate occurrence has been found, performs a naive comparison in order to check if one of the patterns really occurs. The new method we propose is based on the combination of multiple hash functions with the aim of improving the filtering phase, i.e. to reduce the number of candidate occurrences found by the algorithm. We use the technique for the generalization of the well known Wu-Manber algorithm [16] and conduct an experimental evaluation to show the efficiency of the method.

Before entering into details, we need a bit of notations and terminology. A string p of length $m \geqslant 0$ is represented as a finite array $p[0\,..\,m-1]$ of characters from a finite alphabet Σ of constant size σ. By $p[i]$ we denote the $(i+1)$-th character of p, for $0 \leq i < m$. Likewise, by $p[i\,..\,j]$ we denote the substring of p contained between the $(i+1)$-th and the $(j+1)$-th characters of p, for $0 \leq i \leq j < m$. A substring of the form $p[0\,..\,i]$ is called a *prefix* of p and a substring of the form $p[i\,..\,m-1]$ is called a *suffix* of p for $0 \leqslant i \leqslant m-1$.

Given a set of r patterns $P = \{p_0, p_1, \ldots, p_{r-1}\}$ we indicate with symbol m_i the length of the pattern p_i, for $0 \leq i < r$, while the length of the shortest pattern is denoted by m. Finally, we recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` "&" and the `left shift` "$\ll$" operator (shifts to the left its first argument by a number of bits equal to its second argument).

II. Previous Results

The problem of searching DNA sequences has been extensively studied in the last years and its importance in modern biology has led to produce much works. In the field of single string matching, Kalsi *et al.* [9] performed an experimental comparison of the most efficient algorithms for searching biological sequences. In addition in [6], [7] Faro and Lecroq presented an extensive evaluation of (almost) all existing exact string matching algorithms under various conditions, including alphabet of four characters and DNA sequences. Navarro and Raffinot presented a comparison [13] of all matching algorithms on biological sequences, including multiple pattern matching algorithms. More recently, Kouzinopoulos and Margaritis conducted another experimental comparison [11] taking into account the most recent solutions.

978-1-4673-4358-9/12/$31.00 ©2012 IEEE

Basically a string matching algorithm uses a window to scan the text. The size of this window is equal to the minimal length of a pattern in the set of patterns. It first aligns the left ends of the window and the text. Then it checks if any pattern in the set occurs in the window (this specific work is called an *attempt*) and then 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 best algorithms for searching DNA sequences are based on *filtering* methods. Specifically, instead of checking at each position of the text if each pattern in the set occurs, it seems to be more efficient to *filter* text positions and check only if the contents of the window *looks like* any pattern in the set. When a resemblance has been detected a naive check of the occurrence is performed. In order to detect the resemblance between the pattern and the text window efficient algorithms use *bit-parallelism* or *character comparisons*. Both techniques can be improved by using condensed alphabets and hashing.

In particular each pattern p of the set is arranged using a condensed alphabet. In such a representation groups of q adjacent characters of the pattern are condensed in a single character by using a suitable hash function $h : \Sigma^q \rightarrow \{0, \ldots, \text{MAX}\}$, for a constant value MAX. In practice, the value of q varies with m and the size of the alphabet and the value of the constant MAX varies with the memory space available. [1] Thus a pattern p of length m translates in a condensed pattern $p^{(q)}$ of length $m - q + 1$ where $p^{(q)}[i] = h(p[i \,..\, i + q - 1])$, for $0 \leq i \leq m - q$. The hashing method adopted in standard implementations of condensed alphabets is based on a shift-and-addition procedure. Specifically, if $x \in \Sigma^q$, with $x = x[0 \,..\, q - 1]$, then $h(x)$ can be efficiently computed by

$$h(x) = \sum_{i=0}^{q-1} ((x[i] \;\&\; M) \ll k(q - i - 1)) \qquad (1)$$

where k is a constant and M is a bit-mask both dependent on q. In practice k is set to $\lfloor \omega/q \rfloor$ and M is set to $0^{\omega-k}1^k$, where ω is the size of the register used for hashing q-grams. Such computation turns out to be particularly effective when searching on DNA sequences. The DNA alphabet is formed by the four characters {a, c, g, t}, whose ASCII codes are {01000001, 01000011, 01000111, 01010100}. Using $k = 3$ and $M = 00000111$ leads to a perfect hashing. However for larger alphabets or when q is greater than 5 only a resemblance can be used.

The bit-parallelism technique [1] takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to ω, where ω is the number of bits in a computer word. This technique is particularly suitable for simulating non-deterministic automata for a single pattern [1] and for multiple patterns [2], [3]. The Multiple Backward Nondeterministic DAWG Matching algorithm [13] (MBNDM) is included in the MPScan tool [15] and was proved to be the most efficient bit-parallel algorithm for searching DNA sequences in most practical cases. It simulates the suffix automaton of a sequence of classes of characters obtained by superimposing all the patterns in the set. During each attempt the window of the text is scanned from right to left, and when a candidate occurrence is found all the patterns beginning with the first condensed character of the window are checked.

In the field of string matching in biological sequences comparisons of characters provides a simple and efficient method to obtain a sub-linear number of character comparisons in most practical situations. The most efficient result using this approach was presented by Wu and Manber (WM) in [16] for the multiple pattern matching problem. Their algorithm is a generalization of the Boyer-Moore-Horspool algorithm [8] to the multiple pattern cases using condensed alphabets. The idea is to consider groups of q adjacent characters as a single condensed character. Then during the searching phase the algorithms search for candidate occurrences by querying if the rightmost condensed character of the current window of the text appears as the rightmost condensed character of any pattern in the set. In such a case a naive verification is run for each pattern which satisfies the query, otherwise a shift is performed and the next window is processed. Both queries and shifts are computed using a precomputed table which stores the positions of all condensed characters appearing in the patterns.

Both the MBNDM and the WM algorithms just use condensed alphabets with values $q = 5$ and $k = 3$, and turn out to be particularly effective in practice.

Efficient solutions have been also designed for searching on packed DNA sequences [14], [10], [5]. However in this paper we do not take into account this type of solutions since they require a different type of data representation.

III. A Multiple Hashing Solution

When searching for longer patterns and/or larger sets of patterns it is always convenient to use a filtration method which better localize candidate occurrences. This generally translates in involving larger q-grams. In this context the value of q represents a trade-off between the computational time required for computing the q-grams for each window of the text and the computational time needed for checking false positive candidate occurrences. The larger is the value of q, the more time is needed to compute each q-gram. On the other hand, the larger is the value of q, the smaller is the number of false positive the algorithm finds along the text. A similar reasoning can be followed for the value of k involved in the computation of the hash value given in (1). Given a value of q, the larger is the value of k, the smaller is the number of false positive. However the values of q and k strongly depend on the available memory. Practically it is reasonable to consider the hash values computed on 32 (or 64) bits and each key (a q-gram) computed on 8 (or 16) bits. This translates in an extension of the alphabet's dimension up to $65,536$ different values. The consequence is that larger values of k implies smaller values of q, and the opposite. For instance, using 16

[1] In our implementation we use a value of MAX equal to 2^{16} and use a 16-bit register for each hash value.

$q:k$	8	16	32	64	128
3 : 5	167 660	167 714	167 785	167 829	168 791
4 : 4	44.6 184	44.6 210	44.5 225	44.6 231	44.6 225
6 : 2	33.7 146	33.6 155	33.6 166	33.7 170	33.7 169
8 : 2	5.10 47.3	4.97 33.2	4.93 34.3	4.92 37.8	4.92 35.6
10 : 1		119 507	119 539	119 559	119 548
12 : 1		50.3 220	49.6 235	49.6 239	49.8 228
14 : 1		21.4 101	20.6 102	20.4 104	20.3 100
16 : 1		9.15 57.2	8.61 44.2	8.00 43.6	7.68 41.2

TABLE I
COMPARISON OF STANDARD IMPLEMENTATIONS OF THE WM(q) ALGORITHM WHILE SEARCHING $10,000$ PATTERNS ON A GENOME TEXT. FOR EACH LENGTH OF THE PATTERN WE PRESENT (ON THE LEFT) THE NUMBER OF NAIVE VERIFICATIONS PERFORMED FOR EACH TEXT POSITION AND (ON THE RIGHT) THE RUNNING TIMES IN HUNDREDTHS OF SECONDS.

bits for representing a q-gram and 3 bits for each text character, the value of q is up to 5.

Table I shows the number of naive verifications and the corresponding running times of the standard implementations of the WM algorithm using different values of q. For each variant the value of k used for computing the hash value is depicted (the details of the settings are given in Section IV).

First of all notice that the increasing in the value of q (and the corresponding decreasing of k) strongly affects the number of verifications performed during the searching phase. For instance WM(8) uses a value $k = 2$ and performs approximately 5 verifications for each text position while. On the other hand WM(10) decreases the value of k to 1 but increases the number of verifications of almost 20 times.

Notice also that the number of naive verifications strongly influence the corresponding running times. However the running times is also affected by the time needed for computing the condensed character, i.e. by the value of q. For instance while the number of verifications from WM(6) to WM(8) reduces of almost 6 times, the running times between the same variants reduces only of approximately 3 times.

A. The basic idea

Since extending the number of bits used for representing a q-gram is time and space consuming, we propose an alternative solution based on a multiple hashing approach. The idea is straightforward but effective and consists of using multiple hash functions in order to reduce the number of naive verifications and/or to extend the portion of the window involved in the filtration phase.

Our solution reaches this goal without increasing the required memory and the computational time for preprocessing the set of patterns. We propose to use γ different hash functions (or γ different copies of the same hash function) to index different consecutive q-grams in the text. Then a fingerprint of all q-grams is obtained by mixing their hash values through an appropriate function. Specifically, given two constant values $q > 0$ and $\gamma > 0$, and a string x of length $m \geq q\gamma$, we indicate with symbol $g_x(i,q)$ the i-th q-gram of x (proceeding from right to left). More formally we have $g_x(i,q) = x[m-iq\,..\,m-1-(i-1)q]$ for $0 < i \leq \gamma$. Moreover an auxiliary function $mix : \mathbb{N}^\gamma \to \mathbb{N}$ is defined as

$$mix(a_1, a_2, \ldots, a_\gamma) = \sum_{i=1}^{\gamma} (a_i \ll (\gamma - i))$$

where $a_i \in \mathbb{N}$ for all $0 < i \leq \gamma$. Roughly speaking the mix function is used for combining γ different hash values into a single hash value.

In this context a candidate occurrence is located when all q-grams of the current window of the text resemble their counterpart q-grams in any pattern in the set P. In particular, given two strings x and y we say that x resembles y if

$$mix\Big(g_x(1,q), g_x(2,q), \ldots, g_x(\gamma,q)\Big) = \\ = mix\Big(g_y(1,q), g_y(2,q), \ldots, g_y(\gamma,q)\Big).$$

Our proposal naturally arises from the observation that in most practical cases to use two or three combined q-grams turns out to be more convenient than using a single $(2q)$- or $(3q)$-gram, even if, due to the use of the mix function for combining the γ different q-grams in a single hash value, the number of candidate occurrences slightly increases. Experimental results presented in Section IV confirm our assumption.

B. The algorithm

In this section we briefly describe the preprocessing and the searching phase of the algorithm. Figure 1 shows the pseudocode of the multiple pattern matching algorithm based on $\gamma = 2$ hash functions and q-grams with $q = 3$.

The preprocessing phase consists in computing γ different shift values for all possible strings of length q. In particular every substring $x \in \Sigma^q$ is hashed into a value, $h(x)$, which is used to index the shift value in γ shift vectors, $sh_1, sh_2, \ldots, sh_\gamma$, respectively, all of size MAX. Specifically

$$sh_i[c] = \min\Big(\{m-iq+1\} \cup \{\ell \leq m-iq \mid p_j \in P \text{ and } \\ h_i(p_j[m-1-\ell\,..\,m-2-\ell+q]) = c\}\Big)$$

for $1 \leq i \leq \gamma$, $0 \leq c <$MAX and where we recall that m denotes the length of the shortest pattern in P. An additional data structure is maintained in order to collect the set of patterns which resemble with a specific set of q-grams. In particular a vector F of dimension MAX is defined as

$$F[u] = \{p \in P \mid mix\,(g_p(1,q), g_p(2,q), \ldots, g_p(\gamma,q)) = u\}$$

for $0 \leq u <$ MAX.

The searching phase of the algorithm is based on a standard sliding window mechanism. Each attempt (lines 5-13) consists in reading the rightmost γ substrings of length q of the current window of the text, i.e. $s_i = g_p(1,q)$ for $0 < i \leq \gamma$. Then, if $sh_i[h(s_i)] > 0$, for some $0 < i \leq \gamma$, then an advancement of the shift is applied (lines 9 and 13).

An optimal shift advancement should be computed as the $\max\{sh_i[h(s_i)] \mid 0 < i \leq \gamma\}$, however this solution is computationally costly and time consuming. A more efficient solution in practice consists in performing iteratively γ blind

```
HASH3(s, j)
 1.  h ← s[j]&7
 2.  h ← (h ≪ 3) + s[j − 1]&7
 3.  h ← (h ≪ 3) + s[j − 2]&7
 4.  return h

PREPROCESSING(p, m, q)
 1.  for i ← 0 to MAX − 1 do
 2.      F[i] ← null
 3.      sh1[i] ← m − q + 1
 4.      sh2[i] ← m − 2q + 1
 5.  for i ← 0 to r − 1 do
 6.      for j ← 2 to m − 1 do
 7.          h1 ← HASH3(pi, j)
10.          sh1[h1] ← min{sh1[h1], m − 1 − j}
11.      for j ← 2 to m − 1 − q do
12.          h2 ← HASH3(pi, j − 3)
15.          sh2[h2] ← min{sh2[h2], m − 1 − j}
16.      h ← (h1 ≪ 1) + h2
17.      F[h] ← F[h] ∪ {i}
18.  return F

MULTIPLEHASHING(P, r, m, t, n, q)
 1.  F ← Preprocessing(p, m, q)
 2.  t ← t.p0
 3.  j ← m − 1
 4.  while (j < n) do
 5.      do
 6.          h1 ← HASH3(t, j − 3)
 9.          j ← j + sh1[h1]
10.          h2 ← HASH3(t, j − 3)
13.          j ← j + sh2[h2]
14.      while (sh1[h1] > 0 or sh2[h2] > 0)
15.      if j < n then
16.          h ← (h1 ≪ 1) + h2
17.          for each i ∈ F[h] do
18.              if pi = t[j − m + 1 . . j − m + mi]
19.              then output(j − m + 1, i)
20.          j ← j + 1
```

Fig. 1. The Multiple hashing algorithms (using two hash functions and 3-grams) for multiple pattern matching.

shifts of value $sh_i[h(s_i)]$ respectively and to stop only when all advancements turn out to be non-effective (test at line 14).

Observe that the shift advancement $sh_{i+1}[h(s_{i+1})]$ is computed on the new text window aligned after the previous shift $sh_i[h(s_i)]$. Otherwise, when $sh_i[h(s_i)] = 0$ for all $0 < i \leq \gamma$ the patterns in the set $F[mix(h(s_1), h(s_2), \ldots, h(s_\gamma)]$ are examined one by one and naively compared with the current window of the text (lines 17-19). Then an advancement of length 1 is applied (line 20). In addition a copy of the pattern p_0 is attached at the end of the text as a sentinel (line 2) to avoid the current window to pass the right end of the text.

The preprocessing phase of the algorithm requires $\mathcal{O}(\text{MAX}+r)$-space and $\mathcal{O}(\text{MAX}+rmq\gamma)$-time while the searching phase has an $\mathcal{O}(m'n)$ worst case time complexity, where m' is the sum of all patterns lengths in the set P.

IV. Experimental Results

In this section we present experimental evaluations in order to understand the performances of the newly presented algorithm and to compare it against the best algorithm known in literature for multiple pattern matching on DNA sequences.

In particular we tested the MBNDM algorithm used in MP-Scan [15], the WM algorithm [16] and its variants, WM(q, γ), using q-grams and γ hash functions. We use values of q ranging from 2 to 8 and values of γ ranging from 1 to 3.

We implemented the WM variants by using the formula given in (1) and set k to $\lfloor \omega/q \rfloor$ while M has been set to $0^{\omega-k}1^k$, where $\omega = 32$ is the size of the register used for hashing q-grams. All algorithms have been implemented in the C programming language and have been compiled with the GNU C Compiler, using the optimization options -O3. The experiments were executed locally on an MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 4 GB RAM 1333 MHz DDR3, 256 KB of L2 Cache and 6 MB of L3 Cache. Algorithms have been compared in terms of running times, including any preprocessing time, measured with a hardware cycle counter, available on modern CPUs.

For the evaluation we use the genome sequence of $4,638,690$ base pairs of *Escherichia coli*. For the tests on multiple pattern matching we have generated sets of 100, 1000 and $10,000$ patterns of fixed length m. In all cases the patterns were randomly extracted from the text and the value m was made ranging over the values 8, 16, 32, 64 and 128. For each case we reported the mean over the running times of 200 runs.

Tables II, III and IV show experimental results on multiple pattern matching (sets of 100, 1000 and $10,000$ patterns, respectively). Running times are expressed in thousands of seconds. We report the mean of the overall running times (columns with gray background) and the means of the preprocessing and searching times (just on the right). Best times have been boldfaced and underlined. The best searching times among each group of WM(q, γ) variants, with $1 \leq h \leq 3$, have been simply boldfaced.

From our experimental results it turns out that the MBNDM algorithm is the best solution for short patterns and small sets, i.e. $m \leq 16$ and $r \leq 1000$. When the length of the patterns increases the best running times are obtained by the WM(q, γ) algorithms using condensed characters on 8-grams. For small sets of patterns ($r = 100$) the WM$(8, 1)$ algorithm obtains the best results, while for larger sets of pattern the WM$(8, 2)$ and the WM$(8, 3)$ algorithms obtain the best results.

Observe that in most cases the WM$(q, 2\gamma)$ algorithm is faster than the WM$(2q, \gamma)$ algorithm. For instance when searching sets of $10,000$ patterns the WM$(3, 2)$ variant is always 4 times faster than the WM$(6, 1)$ variant, while variant WM$(4, 2)$ is always 2 times faster than WM$(8, 1)$. Similarly in many cases the WM$(q, 3\gamma)$ algorithm is faster than the WM$(3q, \gamma)$ algorithm. This behavior confirms the efficiency of our method, especially for larger sets of patterns. For instance when searching sets of $10,000$ patterns the WM$(2, 3)$ variant is always 3 times faster than the WM$(6, 1)$ variant.

It is important to observe also that the difference in the preprocessing times is negligible when comparing different variants of the WM(q, γ) algorithm. Table V shows the experimental results obtained by comparing the above algorithms in terms of number of naive verifications for each text position on sets of 1000 patterns and $10,000$ patterns. In general the number of verifications reflect the corresponding running times, higher the number of verifications performed during the searching phase, higher the running times of the algorithm.

It is interesting to observe that using two or three hash

m	8			16			32			64			128		
MBNDM	**16.21**	0.20	16.01	**8.71**	0.20	8.51	8.77	0.20	8.57	8.64	0.20	8.44	8.71	0.20	8.51
WM$(2,2)$	**81.88**	0.52	81.36	**81.69**	0.51	81.18	**83.95**	0.52	83.42	**81.81**	0.53	81.28	**82.33**	0.57	81.76
WM$(2,3)$	84.12	0.64	83.48	85.30	0.65	84.65	85.67	0.66	85.01	84.27	0.67	83.60	85.19	0.73	84.45
WM$(3,2)$	**67.47**	0.52	66.95	**67.73**	0.51	67.21	**67.33**	0.52	66.80	**67.52**	0.54	66.98	**67.67**	0.58	67.09
WM$(3,3)$				92.59	0.65	91.95	91.87	0.66	91.21	91.79	0.71	91.08	92.04	0.76	91.27
WM$(4,1)$	**33.43**	0.38	33.05	30.40	0.38	30.02	28.94	0.38	28.57	29.36	0.39	28.97	29.36	0.42	28.95
WM$(4,2)$	35.24	0.53	34.71	**24.87**	0.53	24.33	**23.32**	0.52	22.79	**23.38**	0.55	22.83	**23.38**	0.60	22.78
WM$(4,3)$				26.47	0.66	25.81	23.93	0.66	23.27	23.95	0.71	23.25	23.54	0.74	22.79
WM$(6,1)$	**32.92**	0.37	32.54	20.26	0.37	19.89	16.50	0.37	16.12	15.45	0.39	15.06	14.71	0.39	14.31
WM$(6,2)$				**15.43**	0.52	14.91	11.08	0.52	10.56	10.29	0.56	9.74	9.92	0.59	9.33
WM$(6,3)$							**10.98**	0.66	10.33	**9.99**	0.71	9.28	**9.64**	0.77	8.87
WM$(8,1)$	**50.51**	0.36	50.14	**9.86**	0.37	9.49	**5.95**	0.37	5.58	**4.77**	0.40	4.37	**4.19**	0.43	3.77
WM$(8,2)$				15.61	0.53	15.08	6.79	0.54	6.26	5.13	0.57	4.56	4.46	0.63	3.83
WM$(8,3)$							8.29	0.67	7.61	5.73	0.74	4.99	4.93	0.86	4.08

TABLE II
EXPERIMENTAL RESULTS FOR SEARCHING SET OF 100 PATTERNS ON A GENOME SEQUENCE.

m	8			16			32			64			128		
MBNDM	**85.11**	0.27	84.84	**30.76**	0.29	30.47	30.76	0.31	30.45	31.11	0.33	30.78	31.30	0.35	30.95
WM$(2,2)$	**224.92**	0.63	224.2	**230.01**	0.66	229.3	**240.50**	0.72	239.7	**247.47**	0.91	246.5	**247.68**	1.23	246.4
WM$(2,3)$	129.26	0.78	128.4	128.19	0.82	127.3	128.88	0.94	127.9	132.65	1.21	131.4	133.54	1.70	131.8
WM$(3,2)$	**101.13**	0.62	100.5	**103.12**	0.67	102.4	**102.81**	0.75	102.0	**102.44**	0.94	101.5	**103.98**	1.30	102.6
WM$(3,3)$				113.67	0.84	112.8	114.07	1.02	113.0	113.61	1.35	112.2	114.36	1.85	112.5
WM$(4,1)$	189.21	0.45	188.7	192.03	0.48	191.5	201.14	0.52	200.6	206.16	0.65	205.5	209.00	0.85	208.1
WM$(4,2)$	**95.47**	0.61	94.86	**91.59**	0.69	90.91	**91.55**	0.80	90.74	**91.51**	1.05	90.46	**92.24**	1.51	90.73
WM$(4,3)$				131.81	0.82	130.9	132.33	1.03	131.3	131.56	1.38	130.1	132.66	2.05	130.6
WM$(6,1)$	**155.74**	0.44	155.3	147.53	0.47	147.0	152.15	0.52	151.6	154.20	0.64	153.5	155.51	0.87	154.6
WM$(6,2)$				**72.32**	0.64	71.68	**71.47**	0.76	70.71	**71.37**	1.00	70.37	**71.59**	1.42	70.17
WM$(6,3)$							85.67	0.96	84.70	85.74	1.34	84.40	86.16	2.04	84.13
WM$(8,1)$	**98.58**	0.43	98.14	**40.39**	0.48	39.91	33.16	0.57	32.59	31.16	0.75	30.42	31.01	1.07	29.94
WM$(8,2)$				40.79	0.65	40.14	**22.48**	0.83	21.65	20.43	1.16	19.27	20.50	1.79	18.71
WM$(8,3)$							22.86	1.06	21.79	**19.68**	1.60	18.08	**20.16**	2.71	17.45

TABLE III
EXPERIMENTAL RESULTS FOR SEARCHING SET OF 1000 PATTERNS ON A GENOME SEQUENCE.

m	8			16			32			64			128		
MBNDM	752.32	0.81	751.5	924.18	1.05	923.1	994.26	1.08	993.1	1057.94	1.39	1056	1089.43	1.82	1087
WM$(2,2)$	2063.6	1.26	2062.35	2358.5	1.65	2356.86	2571.7	2.38	2569.36	2681.8	4.07	2677.76	2634.8	7.17	2627.63
WM$(2,3)$	**481.29**	1.47	479.8	**481.97**	2.01	479.9	**516.37**	3.10	513.2	**545.51**	5.72	539.7	**538.88**	10.29	528.5
WM$(3,2)$	**341.06**	1.36	339.6	365.18	1.68	363.5	381.18	2.55	378.6	394.11	4.62	389.4	389.89	8.12	381.7
WM$(3,3)$				**158.99**	2.23	156.7	**158.69**	3.79	154.9	**161.09**	6.79	154.3	**164.62**	12.18	152.4
WM$(4,1)$	1845.5	1.06	1844	2103.1	1.30	2101	2256.6	1.81	2254	2319.1	3.00	2316	2253.6	5.25	2248
WM$(4,2)$	**212.04**	1.28	210.7	185.05	1.83	183.2	188.04	2.99	185.0	192.83	5.48	187.3	195.04	9.82	185.2
WM$(4,3)$				**161.70**	2.23	159.4	**168.89**	4.13	164.7	**166.54**	7.67	158.8	**170.72**	14.11	156.6
WM$(6,1)$	**1463.0**	1.05	1461	1557.6	1.25	1556	1668.0	1.83	1666	1708.5	2.97	1705	1694.0	5.40	1688
WM$(6,2)$				**195.67**	1.53	194.1	203.74	2.65	201.0	218.30	4.96	213.3	213.09	9.37	203.7
WM$(6,3)$							**185.16**	3.63	181.52	**198.35**	7.57	190.78	**195.06**	14.10	180.96
WM$(8,1)$	**473.15**	0.91	472.2	332.39	1.34	331.0	343.12	2.14	340.9	378.56	3.94	374.6	356.40	7.15	349.2
WM$(8,2)$				**135.72**	1.52	134.1	**123.89**	3.23	120.6	**136.73**	6.72	130.0	**136.20**	12.83	123.3
WM$(8,3)$							152.07	4.29	147.7	168.30	10.18	158.1	170.04	20.43	149.6

TABLE IV
EXPERIMENTAL RESULTS FOR SEARCHING SET OF $10,000$ PATTERNS ON A GENOME SEQUENCE.

functions considerably reduces the number of verification calls. For instance when using 3 hash functions on 2-grams on sets of 1000 patterns, the WM(q,γ) algorithm performs approximately 0.7 verifications for each text position, while it performs approximately 3.2 verifications for each text positions when using a single function on 6-grams. The gain is more evident if we consider the number of verifications when using 2 hash functions on 3-grams. In this case the value is approximately 4.7, almost 6 times lower than that obtained by using a single hash function. A similar behavior can be observed also in the case of experimental results obtained on sets of $10,000$ patterns. It is interesting to observe also that in some cases a reduction of the number of verifications does not correspond to a reduction in the searching phase. This is the case, for instance, of the running times of the WM$(8,2)$ and WM$(8,3)$ variants when searching sets of $10,000$ patterns. In fact, while the number of verifications reduces of almost one half, the running times increase. This behavior is due to the time consumed in computing more hash functions when the gain in number of verifications is not significant.

Thus, when searching for small sets of patterns it is convenient to use a single function and large q-grams. Otherwise, when the size of the set of patterns increases, it is convenient to use two hash functions on large q-grams, and in particular $q = 4$ and $\gamma = 2$ for $m < 16$, and $q = 8$ and $\gamma = 2$ for $m \geq 16$. Finally we notice that, in our experimental results,

m	8	16	32	64	128
MBNDM	.5261	.0492	.0504	.0492	.0496
WM$(2,2)$	5.428	5.471	5.476	5.408	5.427
WM$(2,3)$	.7368	.7381	.7381	.7339	.7290
WM$(3,2)$	.4762	.4764	.4809	.4738	.4760
WM$(3,3)$	-	.0413	.0419	.0412	.0413
WM$(4,1)$	4.440	4.450	4.443	4.436	4.446
WM$(4,2)$	**.1092**	.1078	.1076	.1089	.1089
WM$(4,3)$	-	.0239	.0242	.0240	.0241
WM$(6,1)$	3.265	3.233	3.227	3.178	3.124
WM$(6,2)$	-	.0995	.0988	.0978	.0966
WM$(6,3)$	-	-	.0267	.0268	.0271
WM$(8,1)$	.5021	.3258	.2631	.2385	.2247
WM$(8,2)$	-	**.0034**	.0022	.0019	.0018
WM$(8,3)$	-	-	**.0006**	**.0005**	**.0005**

m	8	16	32	64	128
MBNDM	11.94	11.76	11.74	11.74	11.76
WM$(2,2)$	54.61	54.44	54.55	54.59	54.56
WM$(2,3)$	7.379	7.388	7.380	7.385	7.398
WM$(3,2)$	4.785	4.770	4.763	4.782	4.783
WM$(3,3)$		.4145	.4151	.4153	.4147
WM$(4,1)$	44.63	44.61	44.54	44.62	44.63
WM$(4,2)$	**1.130**	1.130	1.134	1.132	1.128
WM$(4,3)$		.2634	.2636	.2631	.2626
WM$(6,1)$	33.75	33.66	33.67	33.75	33.74
WM$(6,2)$		1.527	1.530	1.525	1.522
WM$(6,3)$			.5842	.5834	.5833
WM$(8,1)$	5.105	4.971	4.937	4.929	4.929
WM$(8,2)$		**.2024**	.1912	.1903	.1903
WM$(8,3)$			**.1111**	**.1109**	**.1108**

TABLE V
NUMBER OF NAIVE VERIFICATIONS PERFORMED FOR EACH TEXT POSITION: 1.000 PATTERNS (TOP) AND 10.000 PATTERNS (BOTTOM).

r / m	8	16	32	64	128
100	0.49	0.88	1.47	1.81	2.07
1.000	0.89	0.76	1.36	1.58	1.55
10.000	3.53	6.80	8.02	7.73	7.99

TABLE VI
THE SPEED UPS OBTAINED VIA WM(q,γ) ALGORITHMS.

the choice of values of q larger than 8 and values of γ larger than 3 lead to bad performances.

Finally Table VI summarizes the speed up ratios achieved via the new variants. The values has been obtained by dividing the timing obtained by the MBNDM algorithm by the best timing achieved by WM(q,γ). As can be viewed from that table, the newly proposed solutions are in most cases faster than the MBNDM algorithm. The most significant performance enhancement is achieved on sets of $10,000$ patterns, where up to more than 8 fold increase in speed has been observed. The gain in speed becomes more and more significant with the increasing size of the patterns sets as well as the lengths of the patterns. For example while searching 100 patterns of length 32, the WM(q,γ) algorithm is 1.47 times faster than MBNDM, where that speed up is 2.07 when considering patterns of length 128 each. The gain in speed up increases up to 7.99 when considering sets of $10,000$ pattern of length 128 each.

V. CONCLUSIONS AND PERSPECTIVES

We provide new solutions for the exact multiple pattern matching problem based on multiple hash functions which turn to be very efficient in practice. Although we provided simple implementations of our approach, experimental benchmarks showed that on every set sizes the best solution among the proposed variants is faster in most cases than the MBNDM algorithm, which is considered the faster solution for matching DNA sequences. Considering the orders of magnitude performance gain, the presented technique becomes a strong alternative for multiple exact matching of large sets of patterns on biological sequences. However it would be interesting to investigate different types of hash functions in order to find more efficient combinations. In addition it would be interesting also to compare the performance of the newly presented solution against very recent and efficient solutions [4] in the case of large alphabets.

REFERENCES

[1] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, October 1992.

[2] Domenico Cantone, Simone Faro, and Emanuele Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. In Amihood Amir and Laxmi Parida, editors, *Combinatorial Pattern Matching*, volume 6129 of *Lecture Notes in Computer Science*, pages 288–298. Springer Berlin / Heidelberg, 2010.

[3] Domenico Cantone, Simone Faro, and Emanuele Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. *Inf. Comput.*, 213:3–12, 2012.

[4] Simone Faro and M. Oğuzhan Külekci. Fast multiple string matching using streaming simd extensions technology. In *Proc. of the 19th Intern. Symp. on String Processing and Information Retrieval*, volume 7608 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 2012.

[5] Simone Faro and Thierry Lecroq. An efficient matching algorithm for encoded dna sequences and binary strings. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching*, CPM '09, pages 106–115, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Simone Faro and Thierry Lecroq. The exact string matching problem: a comprehensive experimental evaluation. *CoRR*, abs/1012.2547, 2010.

[7] Simone Faro and Thierry Lecroq. The exact online string matching problem: a review of the most recent results. *ACM Computing Surveys*, 45(2):to appear, 2013.

[8] R. Nigel Horspool. Practical fast searching in strings. *Softw., Pract. Exper.*, 10(6):501–506, 1980.

[9] Petri Kalsi, Hannu Peltola, and Jorma Tarhio. Comparison of exact string matching algorithms for biological sequences. In *BIRD*, pages 417–426, 2008.

[10] Jin Wook Kim, Eunsang Kim, and Kunsoo Park. Fast matching method for dna sequences. In Bo Chen, Mike Paterson, and Guochuan Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, First International Symposium*, volume 4614 of *Lecture Notes in Computer Science*, pages 271–281. Springer, 2007.

[11] Charalampos S. Kouzinopoulos, Panagiotis D. Michailidis, and Konstantinos G. Margaritis. Experimental results on multiple pattern matching algorithms for biological sequences. In *BIOINFORMATICS*, pages 274–277, 2011.

[12] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology, Vol. 10, Nr. 3, pages R25–10, 2009.

[13] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[14] Jussi Rautio, Jani Tanninen, and Jorma Tarhio. String matching with stopper encoding and code splitting. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, pages 42–52, 2002.

[15] Eric Rivals, Leena Salmela, Petteri Kiiskinen, Petri Kalsi, and Jorma Tarhio. Mpscan: fast localisation of multiple reads in genomes. In *Proceedings of the 9th international conference on Algorithms in bioinformatics*, WABI'09, pages 246–260, Springer-Verlag, 2009.

[16] Sun Wu and Udi Manber. A fast algorithm for multi-pattern searching. Technical report, 1994.