Computing and Informatics, Vol. 38, 2019, 937–962, doi: 10.31577/cai_2019_4_937

REVISITING MULTIPLE PATTERN MATCHING

Robert Susik, Szymon Grabowski

Institute of Applied Computer Science, Lodz University of Technology
Al. Politechniki 11, 90 924 Łódź, Poland
e-mail: rsusik@kis.p.lodz.pl, sgrabow@kis.p.lodz.pl

Kimmo Fredriksson

School of Computing, University of Eastern Finland
P.O.B. 1627, FI-70211 Kuopio, Finland
e-mail: kimmo.fredriksson@uef.fi

Abstract. We consider the classical exact multiple string matching problem. The proposed solution is based on a combination of a few ideas: using q-grams instead of single characters, pattern superimposition, bit-parallelism and alphabet size reduction. We discuss the pros and cons of various alternatives to achieve the possibly best combination of techniques. The main contribution of this paper are different alphabet mapping methods that allow to reduce memory requirements and use larger q-grams. The experimental results show that the presented algorithm is competitive in most practical cases. One of the tests shows also that tailoring our scheme to search over a byte-encoded text results in speedups in comparison to searching over a plain text.

Keywords: Text algorithms, pattern matching, multiple pattern matching, q-grams, bit-parallelism, compressed pattern matching

Mathematics Subject Classification 2010: 68W32

1 INTRODUCTION

The problem of multiple pattern matching can be stated as follows: Given text T of length n and pattern set $\mathcal{P} = \{P_0, \ldots, P_{r-1}\}$, in which each pattern is of length m, and all considered sequences are over common alphabet Σ of size σ, find all pattern occurrences in T. The pattern equal length requirement may be removed (although not all algorithms easily handle uneven patterns). Multiple pattern matching is a classic problem, with over 40 years of history and applications in intrusion detection, anti-virus software, spam filtering and bioinformatics, to name a few. As this problem is a straightforward generalization of a single pattern matching, it is perhaps no surprise that many techniques worked out for a single pattern are borrowed for efficient algorithms for multiple patterns.

Our paper presents a novel algorithm for multiple pattern matching, being a careful combination of several known techniques: using q-grams combined with pattern superimposition, bit-parallelism and alphabet size reduction. The experimental results will show that the presented solution, MAG (Multi AOSO on q-Grams), usually dominates over its competitors on texts with diverse characteristics.

1.1 Related Work

A naive approach to searching for r patterns in a text is to use any single pattern search algorithm r times. As the performance grows linearly with r, such a technique cannot be reasonably used when, e.g., r exceeds 10. Non-trivial algorithms for the presented problem can roughly be divided into three different categories, based on the location of pattern substrings they try to find (and then to extend). More specifically, these algorithms are based on

1. prefix searching,
2. suffix searching and
3. factor searching, respectively.

Another taxonomy of the existing solutions classifies them according to whether they are based on character comparisons, hashing, or bit-parallelism. Yet another view is to say that they are based on filtering, aiming for good average case complexity, or on some kind of "direct search" with good worst case complexity guarantees. These different categorizations are of course not mutually exclusive, and many solutions are hybrids that borrow ideas from several techniques. For a good overview of the classical solutions, the reader is referred to, e.g., [27, 19, 10]. We briefly review some of them in the following paragraphs.

Perhaps the most famous solution to the multiple pattern matching problem is the Aho-Corasick (AC) [1] algorithm, which generalizes the Knuth-Morris-Pratt algorithm [22] for a single pattern. The AC algorithm follows the prefix-based approach and it builds a pattern trie with extra (failure) links. One can say that AC

works in linear time. More precisely, however, AC total time is $O(M+n+z)$ for constant alphabet, where M, the sum of pattern lengths, is the preprocessing cost, and z is the total number of pattern occurrences in T. For an integer alphabet, i.e., when $\sigma = n^{O(1)}$, it obtains $O(M + n \log \sigma + z)$ time [12]. Fredriksson and Grabowski [18] showed an average-optimal filtering variant of the classic AC algorithm. They built the AC automaton over superimposed subpatterns, which allows to sample the text characters in regular distances, not to miss any match (i.e., any verification). This algorithm is based on similar ideas as the current work.

Another classic algorithm is Commentz-Walter [8], which generalizes the ideas of Boyer-Moore (BM) algorithm [4] for a single pattern, to solve the multiple pattern matching problem (suffix-based approach). Set Horspool (SH) [15, 27] may be considered its more practical simplification, exactly in the way that Boyer-Moore-Horspool (BMH) [20] is a simplification of the original BM. Set Horspool makes use of a generalized bad character function. The Horspool technique was used in a different way in an earlier algorithm by Wu and Manber (WM) [32]. These methods are based on backward matching over a sliding text window, which is shifted based on some rule, with the hope that many text characters can be skipped altogether.

The first factor-based algorithms were DAWG-match [9] and MultiBDM [11]. Like Commentz-Walter and Set Horspool, they are based on backward matching. However, instead of recognizing the pattern suffixes, they recognize the factors, which effectively means that they work more per window, but in return they are able to make longer shifts of the sliding window, and in fact they obtain optimal average case complexity. At the same time they are linear in the worst case. The drawback is that these algorithms are reasonably complex and not very efficient in practice. A more practical approach is the Set Backward Oracle Matching (SBOM) algorithm [2], which is based on the same idea as MultiBDM, but uses simpler data structures and is very efficient in practice. Yet another variant is the Succinct Backward DAWG Matching algorithm (SBDM) [17], which is practical for huge pattern sets due to replacing the suffix automaton with succinct index. The factor-based algorithms usually lead to average-optimal [25] time complexity of $O(n \log_\sigma(rm)/m)$.

Bit-parallelism can be used to replace the various automata in the methods mentioned earlier, to obtain very simple and yet competitive incarnations of many classical algorithms. The most standard bit-parallel solution for a single pattern is Shift-Or [3]. The idea is to encode (non-deterministic) automaton as a bitvector, i.e., a small integer value, and simulate all the states in parallel using Boolean logic and arithmetic. The result is often the most practical method for the problem, but the drawback is that the scalability is limited by the number of bits in a computer word, although there exist ways to alleviate this problem somewhat, see [28, 7]. Another way that is applicable to huge pattern sets is to combine bit-parallelism with q-grams; our method is also based on this, and we review the idea and related previous work in detail in the next section. Another practical solution based on q-grams (used for a cache-efficient index structure), where a multiple pattern matching algorithm is

applied in intrusion detection software and an antivirus system, is SigMatch [21]. Using q-grams in searches over DNA (for one or multiple patterns) is recommended in the survey by Rivals et al. [29].

In a somewhat different way, parallelism for multiple pattern matching has been obtained with a GPU. In particular, Kouzinopoulos et al. [24] adopted the well-known (and mentioned earlier) AC, SH, SBOM, WM algorithms, as well as a more recent one, SOG [30], to the CUDA programming model, to obtain from about 10- to 30-fold speedup compared to one CPU core. Their test GPU was an Nvidia GTX280, and we can guess that the speedup would be significantly higher on a more modern GPU. The same authors [23] proposed another solution based on hybrid OpenMP/MPI technique, focusing on searching in biological databases.

Some recent work also recognizes the neglected power of the SIMD instructions, which have been available on commodity computers well over a decade. For example, Faro and Külekci [13] make use of the Intel Streaming SIMD Extensions (SSE) technology, which gives wide registers and many special purpose instructions to work with. They develop (among other things) a wsfp (*word-size fingerprint instruction*) operation, based on hardware opcode for computing CRC32 checksums, which computes an α-bit fingerprint from a w-bit register handled as a block of α characters. Similar values are obtained for all α-sized factors of all the patterns in the preprocessing, and wsfp can therefore be used as a simple yet efficient hash-function to identify text blocks that may contain a matching pattern. The same authors [14] presented later a SIMD-based solution for multiple string matching, focusing on searching short patterns ($16 \leq m < 32$) in genome sequences and English texts.

Our paper is organized as follows. Section 2 describes and discusses the two key concepts underlying our work, q-grams and pattern superimposition. Section 3 presents the description of our algorithm, together with its complexity analysis. Section 4 contains experimental results. The last section concludes and points some avenues for pursuing further research.

A preliminary version of our work appeared in Proc. PSC 2014 [31].

2 ON SUPERIMPOSING Q-GRAMS

A q-gram is usually defined as a contiguous substring (factor) of a string comprising q characters, although, noteworthily, non-contiguous q-grams have also been considered [6]. In what follows, q can be considered a small constant ($2, \ldots, 6$ in practice), although we may analyze the optimal value for a given problem instance. We note that q-grams have been widely used in approximate (single and multiple) string matching, where they can be used to obtain fast filtering algorithms based on exact matching of a set of q-grams. Obviously these algorithms work for the exact case as well, as a special case, but they are not within the scope of this paper. Another use (which is not relevant in our case) is to speed up exact matching of a single pattern by treating the q-grams as a superalphabet [16].

In multiple pattern matching q-grams may be combined using an interesting technique called superimposition. Consider a set of patterns $\mathcal{P} = \{P_0, \ldots, P_{r-1}\}$. We form a single pattern P where each position $P[i]$ is no longer a single character, but a *set* of characters, i.e., $P[i] \subseteq \Sigma$. More precisely, $P[i] = \bigcup_j P_j[i]$. Now P can be used as a *filter*: we search candidate text substrings that might contain an occurrence of any of the patterns in $\mathcal{P}$. In other words, if $T[i+j] \in P[j]$, for all $j \in 0, \ldots, m-1$, then $T[i \ldots i+m-1]$ may match with some pattern in $\mathcal{P}$.

Let us present a simple example. Let $r = 2$ and $\mathcal{P} = \{abba, bbac\}$. The superimposed pattern is then $P = \{a,b\}\{b\}\{a,b\}\{a,c\}$, and there are a total of 8 different strings of length 4 that can match with P (and trigger verification). Therefore we immediately notice one of the problems with this approach, i.e., the probability that some text character t matches a pattern character p is no longer $1/\sigma$ (assuming uniform random distribution), it can be up to r/σ. This gets quickly out of hands when the number of patterns r grows.

To make the technique more useful, we first generate a new set of patterns, and then superimpose. The new patterns have the q-grams as the alphabet, which means the new alphabet has size σ^q, and the probability of a false positive candidate will be considerably lower. There are two main approaches: overlapping and non-overlapping q-grams.

Consider first the overlapping q-grams. For each P_i we generate a new pattern such that $P'_i[j] = P_i[j \ldots j+q-1]$, for $j \in 0 \ldots m-q$, that is, each q-gram $P_i[j \ldots j+q-1]$ is treated as a single "super character" in P'_i. Note also that the pattern lengths are decreased from m to $m-q+1$. Taking the previous example, if $\mathcal{P} = \{abba, bbac\}$ and now $q = 2$, the new pattern set is $\mathcal{P}' = \{[ab][bb][ba], [bb][ba][ac]\}$, where we use the brackets to denote the q-grams. The corresponding superimposed pattern is then $P' = \{[ab],[bb]\}\{[bb],[ba]\}\{[ba],[ac]\}$. To be able to search for P', the text must be factored in exactly the same way.

The other possibility is to use non-overlapping q-grams. In this case we have $P'_i[j] = P_i[(j-1)q+1 \ldots jq]$, for $j \in 0 \ldots \lfloor m/q \rfloor - 1$, and for our running example we get $P' = \{[ab],[bb]\}\{[ba],[ac]\}$. Again, the text must be factored similarly. But the problem now is that only every q^{th} text position is considered, and to solve this problem we must consider all q possible shifts of the original patterns. That is, given a pattern P_i, we generate a set $\hat{P}_i = \{P_i[0 \ldots m-1], P_i[1 \ldots m-1], \ldots, P_i[q-1 \ldots m-1]\}$, and then generate $\hat{P}'_i$, and finally superimpose them.

The two alternatives given above both have some benefits and drawbacks. For overlapping q-grams we have:

- pattern length is large $(m-q+1)$, which implies fewer verifications,
- text length is practically unaffected $(n-q+1)$.

On the other hand, for the non-overlapping ones:

- pattern length is short (m/q), which means potentially more verifications, but bit-parallelism works for bigger m,
- text is shorter too (n/q),

- more patterns to superimpose (factor of q).

In the end, the benefits and drawbacks between the two approaches mostly cancel out each other, except bit-parallelism remains more applicable to non-overlapping q-grams.

To illustrate the power of this technique, let us have, for example, a random text over an alphabet of size $\sigma = 16$ and patterns generated according to the same probability distribution; q-grams are not used yet (i.e., we assume $q = 1$). If $r = 16$, then the expected size of a character class in the superimposed pattern is about 10.3, which means that a match probability for a single character position is about 64 %. Even if high, this value may yet be feasible for long enough patterns, but if we increase r to 64, the character class expected size grows to over 15.7 and the corresponding probability to over 98 %. This implies that match verifications are likely to be invoked for most positions of the text. Using q-grams has the effect of artificially growing the alphabet. In our example, if we use $q = 2$ and thus $\sigma' = 16^2 = 256$, the corresponding probabilities for $r = 16$ and $r = 64$ become about 6 % and 22 %, respectively, so they are significantly lower.

The main problem that remains is to decide between the two choices, properly choose a suitable q, and finally find a good algorithm to search the superimposed pattern. To this end, Salmela et al. [30] presented three algorithms combining the known mechanisms: Shift-Or, BNDM [26] and BMH, with overlapping q-grams; the former two of these algorithms are bit-parallel ones. The resulting algorithms were called SOG, BG and HG, respectively. In general, larger q means better filtering, but on the other hand the size of the data structures (tables) that the algorithms use is $O(\sigma^q)$, which can be prohibitive. BGqus (BG with q-grams, Unrolling and s-bit shift hash method) [33] tries to solve the problem by combining BG with hashing.

Actually, not many classic algorithms can be generalized to handle superimposed patterns (character classes) efficiently, but bit-parallel methods generalize trivially. In the next section we describe our choice, FAOSO [18].

3 OUR ALGORITHM

In [18] a general technique of how to skip text characters, with any (linear time) string matching algorithm that can search for multiple patterns simultaneously was presented, alongside with several applications to known algorithms. In the following we review the idea, and for the moment assume that we already have done all factoring to q-grams, and that we have only a single pattern.

3.1 Average-Optimal Character Skipping

The method takes a parameter k, and from the original pattern generates a set $\mathcal{K}$ of k new patterns $\mathcal{K} = \{P^0, \ldots, P^{k-1}\}$, each of length $m' = \lfloor m/k \rfloor$, as follows:

$$P^j[i] = P[j + ik], \quad j = 0 \ldots k-1, i = 0 \ldots \lfloor m/k \rfloor - 1.$$

In other words, k different alignments of the original pattern P is generated, each alignment containing only every k^{th} character. The total length of the patterns P^j is $k\lfloor m/k \rfloor \leq m$.

Assume now that P occurs at $T[i \ldots i + m - 1]$. From the definition of P^j it directly follows that

$$P^j[h] = T[i + j + hk], \quad j = i \mod k, h = 0 \ldots m' - 1.$$

This means that the set $\mathcal{K}$ can be used as a filter for the pattern P, and that the filter needs only to scan every k^{th} character of T. Figure 1 serves as an illustration.

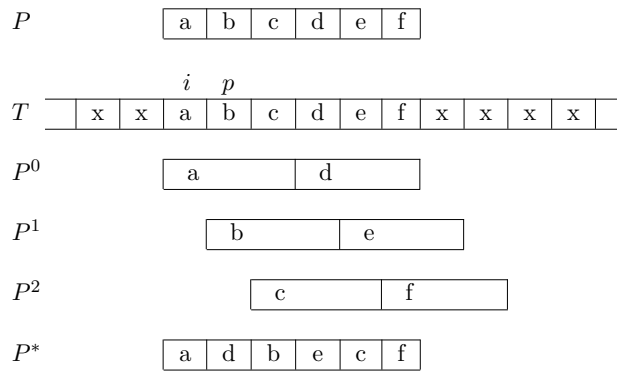

Figure 1. An example. Assume that $P =$ `abcdef` occurs at text position $T[i \ldots i+m-1]$, and that $k = 3$. The current text position is $p = 10$, and $T[p] =$ `b`. The next character the algorithm reads is $T[p+k] = T[13] =$ `e`. This triggers a match of $P^{p \mod k} = P^1$, and the text area $T[p-1 \ldots p-1+m-1] = T[i \ldots i+m-1]$ is verified.

The occurrences of the patterns in $\mathcal{K}$ can be searched for simultaneously using any multiple string matching algorithm. Assuming that the selected string matching algorithm runs generally in $O(n)$ time, then the filtering time becomes $O(n/k)$, as only every k^{th} symbol of T is read. The filter searches for the exact matches of k patterns, each of length $\lfloor m/k \rfloor$. Assuming that each character occurs with probability $1/\sigma$, the probability that P^j occurs (triggering a verification) in a given text position is $(1/\sigma)^{\lfloor m/k \rfloor}$. A brute force verification cost is in the worst case $O(m)$. To keep the total time $O(n/k)$ on average, we select k so that $nm/\sigma^{m/k} = O(n/k)$. This is satisfied for $k = m/(2\log_\sigma(m))$, where the verification cost becomes $O(n/m)$

and filtering cost $O(n \log_\sigma(m)/m)$. The total average time is then dominated by the filtering time, i.e., $O(n \log_\sigma(m)/m)$, which is optimal [34].

3.2 Multiple Matching with q-Grams

To apply the previous idea to multiple matching, we just assume that the (single) input pattern (for the filter) is the non-overlapping q-gram factored and superimposed pattern set. The verification phase just needs to be aware that there are possibly more than one pattern to verify. The analysis remains essentially the same: now the text length is n/q, pattern lengths are m/q, there are r patterns to verify, and the probability of a match is p instead of $1/\sigma$, where $p = O(1 - (1 - (1/\sigma^q))^{qr}) = O((qr)/\sigma^q)$. That is, the filtering time is $O(qn/(kq)) = O(n/k)$, verification cost is $O(rqm)$, and its probability is $O(p^{\lfloor m/(kq) \rfloor})$ for each of the n/q text positions. However, now we have two parameters to optimize, k and q, and the optimal value of one depends on the other.

In practice we want to choose q first, such that the verification probability is as low as possible. This means maximizing q, but the preprocessing cost (and space) grows as $O(\sigma^q)$, and we do not want this to exceed $O(rm)$ (or the filtering cost for that matter). So we select $q = \log_\sigma(rm)$, and then choose k as large as possible. Repeating the above analysis gives then

$$k = O\left(\frac{m}{\log_\sigma(rm)} \cdot \frac{\log_\sigma 1/\rho}{\log_\sigma(rm) + \log_\sigma 1/\rho}\right) \tag{1}$$

where $\rho = \log_\sigma(rm)/m$. We note that this is not average-optimal anymore, although we are still able to skip text characters.

To search the superimposed pattern, we use FAOSO [18], which is based on Shift-Or. The fact that the pattern consists of character classes is not a problem for bit-parallel algorithms, since it only affects the initial preprocessing of a single table. For details see [18]. The filter implemented with FAOSO runs in $O(n/k \cdot \lceil (m/q)/w \rceil)$ time in our case, where w is the number of bits in computer word (typically 64).

We note that Salmela et al. [30] have tried a similar approach, but abandoned it early because it did not look promising for short patterns in their tests.

3.2.1 Implementation

The q-gram, i.e., the super character, must have some suitable representation, and the convenient way is to compute a numerical value in the range $0 \ldots \sigma^q - 1$, which is done as $\sum_{i=0}^{q-1} S[i] \cdot \sigma^i$ for a q-gram $S[0 \ldots q-1]$. This is computed using Horner's method to avoid the exponentiation. We have experimented with two different variants. The first encodes the whole text prior to starting the actual search algorithm, which is then more streamlined. This also means that the total complexity is $\Omega(n)$, the time to encode the text. We call the resulting algorithm SMAG (short of Simple Multi AOSO on q-grams). The other alternative is to keep the text intact, and

compute the numerical representation of the q-gram requested on the fly. This adds just constant overhead to the total complexity. We call this variant MAG (short of Multi AOSO on q-grams). We have verified experimentally that MAG is generally better than SMAG.

3.3 Alphabet Mapping

If the alphabet is large, then selecting a suitable q may become a problem. The reason is that some value q' may be too small to facilitate good filtering capability, yet, using $q = q' + 1$ can be problematic, as the preprocessing time and space grow with σ^q (note that q must be an integer). The other view of using strings of length q as super characters is that we may then say that our characters have $q \log_2 \sigma$ bits, and we want to have more control of how many bits we use. One way to achieve this is to reduce the original alphabet size σ.

We note that in theory this method cannot achieve much, as reducing the alphabet size generally only worsens the filtering capability and therefore forces larger q, but in practice this allows better fine tuning of the parameters.

3.3.1 Histogram Based Alphabet Mapping (HAM)

What we do is that we select some $\sigma' < \sigma$, compute a mapping $\mu : \Sigma \mapsto 0 \ldots \sigma' - 1$, and use $\mu(c)$ whenever the (filtering) algorithm needs to access some character c from the text or the pattern set. Verifications still obviously use the original alphabet. A simple method to achieve this is to compute the histogram of character distribution of the pattern set, and assign code 0 to the most frequent character, 1 to second most frequent, and so on, and put the $\sigma' - 1 \ldots \sigma - 1$ most frequent characters to the last bin, i.e., giving them code $\sigma' - 1$. The text characters not appearing in the patterns also will have code $\sigma' - 1$.

A better strategy is to try to distribute the original characters into σ' bins so that each bin will have (approximately) equal weight, i.e., each $\mu(c)$, where $c \in 0 \ldots \sigma' - 1$ will have (approximately) equal probability of appearance. This is NP-hard optimization problem, so we use a simple greedy heuristic which can be described in few steps:

1. compute the symbol frequencies on the pattern set (using, e.g., hashing to avoid possibly large tables);
2. choose some suitable σ', the size of the mapped alphabet;
3. use method of choice (e.g., bin-packing) to reduce the number of symbols, i.e., map them to range $0 \ldots \sigma' - 1$;
4. optionally use hashing to store the mapping.

3.3.2 Histogram Based Alphabet Mapping on q-Grams (HAMq)

The HAM method can be applied also to q-grams. This variant allows control of table size, and with combined hashing it can accommodate very large q as well. In this case the q-grams are used instead of the alphabet symbols, but the whole process is very much the same. The only issue here is the need of additional hashing to store mapping, along with the corresponding bitvectors needed by FAOSO. We did not implement this method in this paper.

3.3.3 Combined Alphabet Mapping and q-Gram Generation (CAMq)

Yet another method to reduce the alphabet is to combine the q-gram computations with some bit magic. The benefit is that the mapping tables need not to be preprocessed, and this allows further optimizations as we will see shortly. The drawback is that the quality of the mapping is worse than what is achieved with approaches like bin-packing.

Consider a (text sub-)string $S[0 \dots q-1]$ over alphabet Σ of size σ. A simple way to reduce the alphabet is to consider only the ℓ low-order bits of each $S[i]$, where $\ell < \log_2 \sigma$. We can then compute $(q\ell)$-bit q-gram s simply as

$$\begin{aligned} s = (S[0] \,\&\, b) + (S[1] \,\&\, b) << \ell + (S[2] \,\&\, b) \\ << 2\ell + \cdots + (S[q-1] \,\&\, b) << (q-1)\ell \end{aligned} \tag{2}$$

where $b = (1 << \ell) - 1$ and $<<$ denotes the left shift and & the bitwise and.

There is a possibility to have four unique values in ASCII DNA alphabet by right shift (CAMq(dna)).

$$\begin{aligned} s = ((S[0] >> 1) \,\&\, b) + ((S[1] >> 1) \,\&\, b) \\ << \ell + \cdots + ((S[q-1] >> 1) \,\&\, b) << (q-1)\ell. \end{aligned} \tag{3}$$

The main benefit of this approach is that a sequence of shifts and adds can be often replaced by a multiplication (which can be seen as an algorithm performing just that). As an illustrative example, consider the case $\ell = 2$ and hence $b = 3$ (which coincides to DNA nicely). As an implementation detail, assume that the text is 8-bit ASCII text, and it is possible to address the text, a sequence of characters, as a sequence of 32-bit integers (which is easy, e.g., in C). Then to compute a 8-bit 4-gram s we can simply apply the transform:

$$s = (((x >> 1) \,\&\, 0\text{x}03030303) * 0\text{x}40100401) >> 24 \tag{4}$$

where x is the 32-bit integer containing $S[0 \dots 3]$. Assuming 4-letter DNA alphabet, with a right shift (by 1) and the (parallel) masking we obtain unique (and case insensitive) 2-bit codes for all four characters. If the alphabet is larger (many DNA sequences have rare extra symbols), those will be mapped into the same range,

0 . . . 3. The multiplication then shifts and adds all those codes into an 8-bit value, and the final shift moves the 4-gram to the lower bits (CAMq(opt)). Larger q-grams can be obtained by repeating the code.

3.4 Preprocessing and Verification

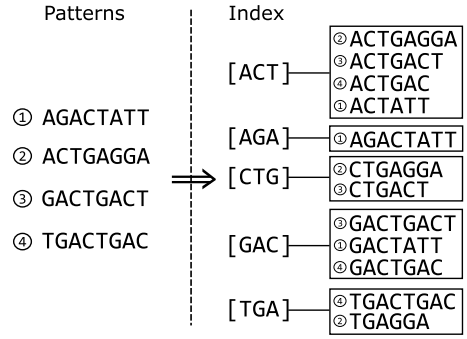

Figure 2. An example of the data structure used for verification; $r = 4$, $q = 3$

The algorithm only returns the positions of possible matches so each of the position needs to be verified. There is no information if the string on returned position belongs to the list of the patterns or not so the verification is necessary. To verify which (if any) of the patterns is found on a given position, the algorithm maintains a dictionary with q-grams as keys and lists of the patterns which contain the key as a substring starting at position $0, \ldots, q-1$ as the associated values. The total length of all the lists stored in the dictionary is rq.

Figure 2 presents a simple example of the described dictionary. The patterns are shown on the left and the dictionary (I) of mappings from 3-grams to patterns containing them (only their respective suffixes are presented) are given on the right. For example, the second element on the list I[“CTG”] is 3 because “CTG” is the third q-gram in the 3rd pattern. The pattern occurs in two other lists: I[“GAC”] and I[“ACT”]. The pseudocode of building such structure is presented as Algorithm 1.

The presented structure is later used in verification process whose pseudocode is presented as Algorithm 2. After a tentative match at position *pos* is reported, the list of associated patterns, $I[get\_q\_gram(T[pos \ldots pos + q - 1])]$, is searched with the binary search (line 3). The *bsearch* function searches over $I[idx]$ with the *pat* field as the key and returns the maximum range of indexes (i, j) such that for all $i \leq a \leq j$, $T[pos - I[idx][a].off \ldots pos - I[idx][a].off + m - 1] = I[idx][a].pat$.

3.5 Algorithm

The combination of all the described techniques yields a fast algorithm for multiple pattern matching, presented as Algorithm 3. The search phase is based on FAOSO

Algorithm 1 MAG_Build_Index

```
1: function MAG_Build_Index(P)
2:     for j ← 0 ... r − 1 do
3:         for i ← 0 ... q − 1 do
4:             idx ← get_q_gram(P_j[i ... q + i − 1])
5:             el.off ← i
6:             el.pat ← P_j
7:             I[idx] ← I[idx] ∪ {el}
8:         for i ← 0 ... | I | do
9:             sort(I[i], key = pat)                    ▷ Sort by pat
10:    return I
```

Algorithm 2 MAG_Verification

```
1: function MAG_Verification(T, pos, I)
2:     idx ← get_q_gram(T[pos ... pos + q − 1])
3:     [i, j] ← bsearch(I[idx])
4:     for el ← I[idx][i] ... I[idx][j] do
5:         report match at pos − el.off
```

(an algorithm for searching a single pattern) but the preprocessing (lines 3–16) and verification (Algorithm 2) are novel. The preprocessing involves multiple patterns (there are r of them) that consist of character classes built on top of q-grams. There is a possibility that the found position is aligned with one of the first q q-grams and this is why such shifts have been taken into account (line 11). Note that the pseudocode presents the variant with combined alphabet mappings and q-gram generation called *get_q_gram*.

3.6 Searching in Compressed Text

Text compression is a technique widely used to decrease the amount of memory needed to store (or transmit) textual data. Some text compression methods allow to search directly in the compressed text, without prior decompression, which is not only elegant but may also improve the search speed, even compared to searching over non-compressed text. In this section, we adapt our solution to searching in End-Tagged Dense Code (ETDC) [5] compressed text, where the ETDC codes are assigned to words. ETDC is a variable-length byte code in which 7 bits per byte store data and 1 bit (by convention, the highest one) is set in the last byte of each codeword and unset otherwise. The data bits are used fully, which means that (in accordance to the golden rule of data compression, which assigns shorter codewords to more frequent symbols) 1-byte codewords are assigned to 128 most frequent words in the text, 2-byte codewords are assigned to $2^{14} = 16\,384$ next most frequent words, and so on (in practice, for unilingual texts it is enough to maintain at most 3-byte

Algorithm 3 Multi AOSO on q-Grams

```
1:  function MAG(P, T[0...n-1])
2:      I ← MAG_Build_Index(P)
3:      for i ← 0...σ^q − 1 do b[i] = ∼0
4:      m' ← (⌊m/q⌋) − (1 − ⌊(m mod q)/(q − 1)⌋)
5:      j ← 0
6:      h ← 0
7:      mm ← 0
8:      for j ← 0...k − 1 do
9:          for i ← 0...m'/k − 1 do
10:             for z ← 0...r − 1 do
11:                 for o ← 0...q − 1 do
12:                     b[get_q_gram(P_z[ikq̇ + jq̇ + o])] &= ∼(1 << h)
13:         for l ← 0...U − 1 do
14:             mm ← mm | (1 << (h − 1))
15:             h ← h + 1
16:         h ← h − 1
17:     /* Search superimposed pattern in T using FAOSO and verify each
            tentative match at position pos with MAG_Verification(T, pos, I) */
```

codewords). The flag bits not only make the code a prefix one, but also allow instant synchronization in an ETDC stream, which in turn conveniently enables to plug in any character-skipping pattern searching algorithms. Note also that such encoding of a text perceived as a sequence of words is intended to make the text shorter (to about 35 % of its original length in practice), yet the encoded pattern (a phrase consisting of whole words) also tends to be shorter, which mitigates the expected search speedup.

3.6.1 Implementation

We encode the input text with ETDC and build a helper array A which allows us to find the position of the pattern in the original (non-compressed) text. More precisely, before the encoding, the original text T is transformed to a simpler form T' by removing all commas, tabulation symbols, excessive spaces and EOL characters (a period is treated as a separate word). After the ETDC encoding we obtain three streams:

1. the encoded text $E(T')$,
2. the dictionary, which maps the distinct words to their corresponding variable-length codewords, and
3. the array A.

Assume that the length of T' is n' (characters, and also bytes). The array A is of length n'/h (integers), where $h > 1$ is a parameter; larger h produces a smaller array yet reporting a match position is more time consuming. Each $A[i]$, $0 \leq i < n'$, stores a value j such that $T'[j]$ is the first symbol of the word whose codeword in $E(T')$ contains the byte $T'[ih]$. We assume that h is at least the maximum number of bytes per codeword (in practice, it is much larger, otherwise the overhead of A would be significant).

Searching in $E(T')$ is very similar to searching in the plain text T. We first run the MAG code for the encoded patterns over $E(T')$, to obtain their positions in the encoded text. Then, in a postprocessing, we map the found positions onto the original positions in T'.

The mapping, making use of the array A, will be shown on an example. Let us have a short text (extracted from `english.200MB`) $T' =$ "the fire on the hearth filled the chamber ." and a pattern $P =$ "the hearth" (for clarity we use only one pattern, but it easily generalizes to multiple patterns). Figure 3 presents the (simplified) text T', the encoded text $E(T')$ and the helper table A. The goal is to obtain all positions of pattern P in text T' from the found occurrences of $E(P)$ in $E(T')$.

Let us assume that $E(P)$ is the 3-byte sequence $[(128)(36, 189)]$ and let $h = 4$ (note that $|A| = \lceil |E(T')|/h \rceil = \lceil 13/4 \rceil = 4$). The first (and only in the example) occurrence of $E(P)$ in $E(T')$ is at position $pos = 4$. As $pos \bmod h = 0$, we have that $A[\lfloor pos/h \rfloor]$ is aligned with the match position and the returned value $A[\lfloor pos/h \rfloor] = A[1] = 12$ is the match position of P in T'.

In the second case, i.e., when $pos \bmod h \neq 0$, things are slighly more complicated. Again, let us use an example; this time we look for $P =$ "chamber". It is encoded as $E(P) = [(85, 138)(129)]$. Its (first and only) occurrence in $E(T')$ is at position $pos = 10$. As $pos \bmod h \neq 0$, to the largest value of A sampling $E(T')$ *before* pos, that is, $A[\lfloor pos/h \rfloor] = A[2] = 23$, we add the decoded lengths of all the words before the found pattern, starting with the word aligned with $A[\lfloor pos/h \rfloor]$. In our example, there are two such words, spanning $E(T[7 \ldots 9])$ and decoded to "filled the " (note the blank space following the last word), of length 11. Therefore, the returned position of P in T' is $23 + 11 = 34$. Note that the ETDC dictionary is not shown in Figure 3.

```
                  012345678901234567890123456789012345678901234567890123456789012
              T' = the fire on the hearth filled the chamber .
                   0    1  2    3    4    5  6    7  8    9    10 11   12
          E(T') = (128)(112,130)(150)(128) (36,189)   (66,133) (128)  (85,138) (129)
              A =  0              12               23                   42
```

Figure 3. The excerpt from an ETDC-encoded text with the helper array ($h = 4$)

4 EXPERIMENTAL RESULTS

In order to evaluate the performance of our approach, we ran several experiments, using 200 MB versions of selected datasets (`dna`, `english` and `proteins`) from the widely used Pizza & Chili corpus (`http://pizzachili.dcc.uchile.cl/`). MAG variants were implemented in C++ and compiled with `g++` 4.8.1 with `-O3` optimization. The experiments were run on a desktop PC with an Intel i3-2100 CPU clocked at 3.1 GHz with 128 KB L1, 512 KB L2 and 3 MB L3 cache, sporting 4 GB of 1 333 MHz DDR3 RAM and running Ubuntu 13.04 64-bit OS with kernel 3.11.0-17. The MAG source codes can be found at the URLs given below:

1. MAG – `https://github.com/rsusik/mag`,
2. MAG on ETDC – `https://github.com/rsusik/magetdc`.

4.1 Multi AOSO on q-Grams

In this section we compare several variants (cf. Section 3.3.1) of our solution. We use the following naming convention:

- `_dna`, adapted for the `dna` alphabet (each symbol is right shifted),
- `_opt`, an optimized variant (shifts and adds replaced by a multiplication),
- `_l`x, where x is the value of the ℓ parameter ($\sigma = 2^\ell$).

We have nine variants in total:

- `mag` – HAM (Section 3.3.1),
- `mag_l2` – CAMq (Equation (2)), with $\ell = 2$,
- `mag_l3` – as above, $\ell = 3$,
- `mag_l4` – as above, $\ell = 4$,
- `mag_dna_l2` – CAMq(dna) (Equation (3)), $\ell = 2$,
- `mag_dna_l3` – as above, $\ell = 3$,
- `mag_dna_l4` – as above, $\ell = 4$,
- `mag_dna_opt_l2` – CAMq(opt) (Equation (4)), with $\ell = 2$,
- `mag_dna_opt_l3` – as above, $\ell = 3$.

We call CAMq and CAMq(dna) as the generic variants, and CAMq(opt) as the optimized variant (shifts and adds replaced by a multiplication). There are a few parameters to set for the algorithms such as q-gram size, FAOSO striding parameter (k) and the quantized alphabet size σ' (only for HAM). We tested HAM with multiple σ' values ($\{4, 5, \ldots, 26\}$) and experimentally chose only a few later used in the tests, namely $\sigma' = 5$ for `dna`, $\sigma' = 14$ for `english` and `proteins` ($r \in \{10, 100\}$) and $\sigma' = 21$ for `proteins` ($r \in \{1\,000, 10\,000\}$). For all variants we set $q = \min(10, \max(2, \lfloor \log_{\sigma'}(rm) \rceil))$ and the FAOSO parameter k as mentioned

in Equation (1), but rounded to the nearest value from $\{1, 2, 4\}$, and the pattern length as follows:

$$m' = \begin{cases} \lfloor m/q \rfloor - 1, & \text{if } m \bmod q < q - 1, \\ \lfloor m/q \rfloor, & \text{otherwise.} \end{cases}$$

It is possible to tune the parameters for a certain dataset and achieve better results (because the formulas on q and k are designed to be optimal for random text). The RAM usage of our implementation can be roughly expressed as $16\sigma'^q + 24r + rm$ bytes, where 16 and 24 are the internal data structure sizes (in bytes). The variants other than HAM are less flexible in terms of σ', but, on the other hand, do not have the preprocessing phase and an additional array to store the alphabet mapping. We tested both methods generic and optimized (shifts and adds replaced by a multiplication). The tests were performed on a 64-bit machine, so there was a limitation for the maximum alphabet size and the value of q. For example, if $\ell = 2$, then we can (directly) use $q \leq 5$, therefore to deal with a larger q, we need to combine two smaller q-grams, e.g., take $S[0 \ldots 2]S[3 \ldots 5]$ to obtain a 6-gram.

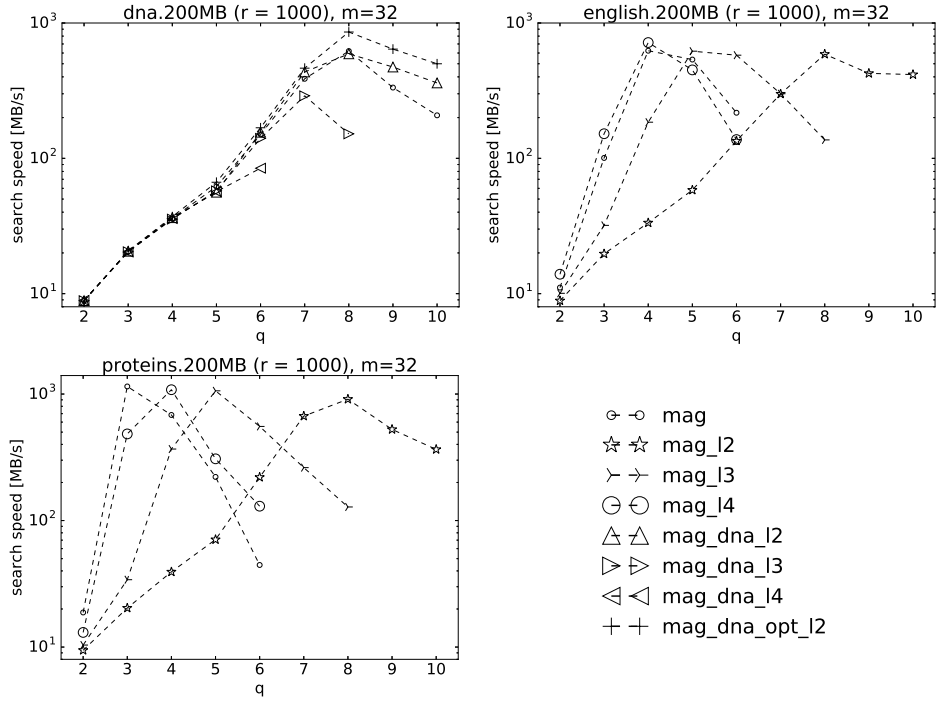

Figure 4. Search speeds for 1 000 patterns, $m = 32$, in function of q

Figure 4 presents search speeds for a collection of $r = 1\,000$ patterns of length $m = 32$ of the MAG algorithm variants using different alphabet mappings, a histogram based mapping (`mag`), and a combined mapping. The main advantage (and disadvantage) of HAM variant is the σ' flexibility that allows use of quite a large range of alphabet mapping. This parameter lets us adapt the HAM (`mag`) in terms of performance to certain dataset or amount of RAM. On the other hand, it forces us to find the most suitable configuration, what is quite tricky and highly depends on the text characteristic. We removed some variants from charts for clarity. The `CAMq(dna)` and `CAMq(opt)` variants were removed for `proteins` and `english` as the results were worse or similar to `CAMq`. The `mag_dna_opt_l3` was removed because the search speed was almost the same as `mag_dna_l3`. The next thing that may be noticed is the difference between larger and smaller alphabets. On `dna` the benefits in performance are visible for a higher q than on the `english` and `proteins` datasets. Another parameter whose optimal value may vary for different alphabet sizes is ℓ. As expected, a larger value of ℓ value is beneficial for larger alphabets (`english` and `proteins`). Note also that there is no point in setting ℓ above 2 for `dna` as its alphabet size is small (four symbols). Finally, the `dna` adapted variant (right shift) proved successful.

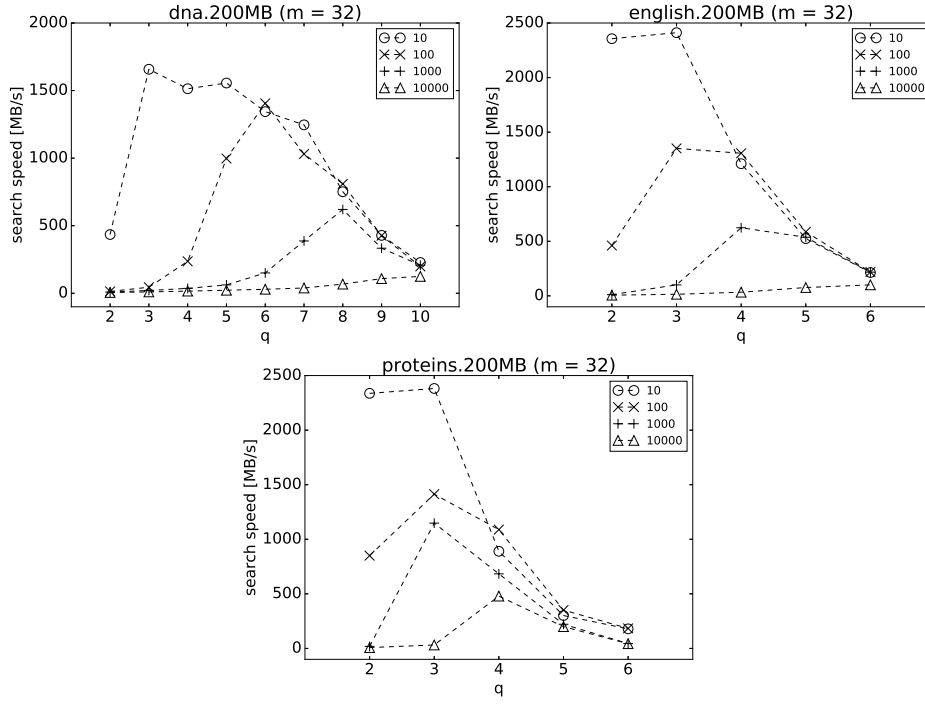

Figure 5. Search speeds for the pattern length $m = 32$ and varying q

We also show how MAG performance changes with growing q for different number of patterns (Figure 5). The figure presents HAM variant for different $r = \{10, 100, 1\,000, 10\,000\}$ as a separate function for multiple values of q values ($q \in \{2, 3, \ldots, 10\}$ for `dna` and $q \in \{2, 3, \ldots, 6\}$ for `english` and `proteins`). As expected, larger q makes sense for increasing number of patterns (r), but a too large value of it slows down the search, presumably due to many cache misses. If the alphabet size is small (such as `dna`) the optimal q-gram size is larger than for bigger alphabet (such as `english` and `proteins`). Note that due to the quantization the original alphabet size does not (significantly) affect the choice of q, but may affect the choice of optimal σ' as it is expected to be smaller than σ.

4.2 Other Solutions

After testing our variants we decided to compare them to other relevant algorithms from the literature, namely:

- BNDM on q-grams (BG) [30],
- Shift-Or on q-grams (SOG) [30],
- BMH on q-grams (HG) [30],
- Rabin-Karp combined with binary search and two-level hashing (RK) [30],
- Multibom and Multibsom, variants of Set Backward Oracle Matching [2],
- Succinct Backward DAWG Matching (SBDM) [17],
- Multi AOSO on q-grams (MAG) (this work).

We tested the competitors' algorithms for a variety of parameters and chose the fastest ones. Namely, there were executed: four variants of BG (using 2-grams, 3-grams, 3-grams with 4 subsets, 3-grams with 2 subsets), two variants of RK (with default parameters and second level hashing), two variants of HG (with default settings and 3-grams), three variants of SOG (2-grams, 2-grams AOSO and 3-grams), and one variant of SBDM ("-A -B -F 1", rank_g and CUSTSIGMA for `english`; some other configurations were excluded after preliminary experiments), Multibom and Multibsom (both with default parameters). All presented results include preprocessing and search times.

In Figure 6 we show the results of all the listed algorithms on `english`, with a fixed pattern length m and growing number of patterns r. The used pattern lengths (one for each plot) are $\{8, 16, 32, 64\}$. Note that some algorithms (or rather their available implementations) cannot handle longer patterns ($m = 64$). To make the chart clear we chose only two variants of our solution that seem to be the most interesting in this case, which are `mag` and `mag_l4`. The `mag_l4` dominates for longer patterns (32, 64) and its performance is mixed for $m = 8$ and $m = 16$. For short ($m = 8, 16$) and many ($r = 10\,000$) patterns SBDM achieves the best results. As expected, for all algorithms the search speed deteriorates with a growing number of patterns, and for $r = 10\,000$ and relatively long patterns ($m = 32$) only MAG

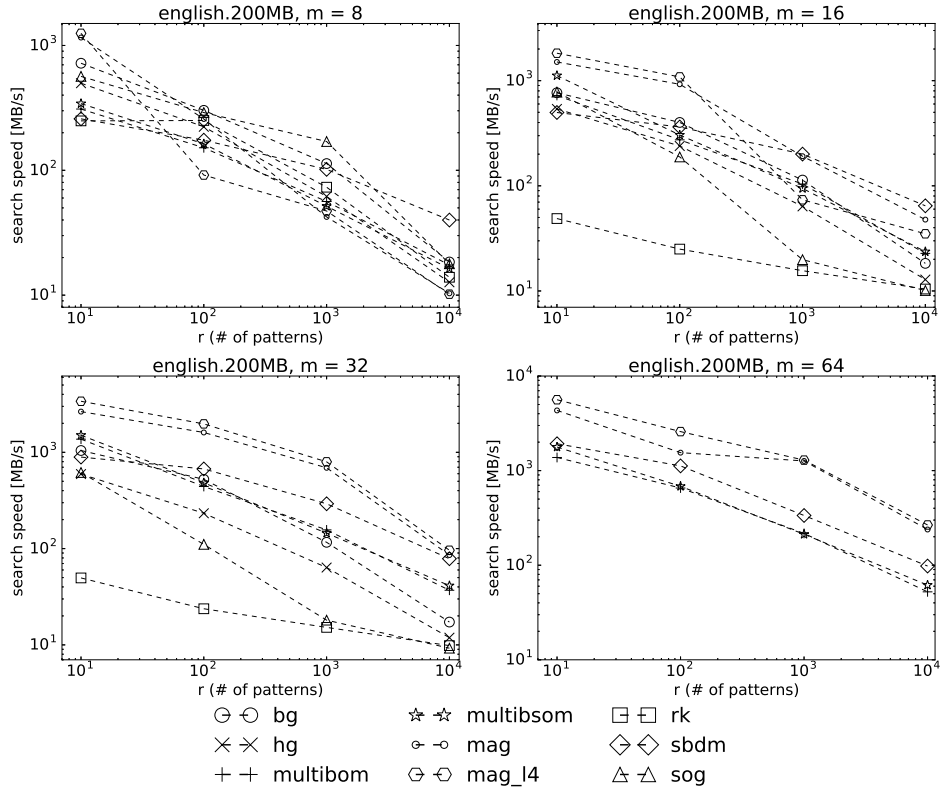

Figure 6. Search speeds for varying number of patterns $r = \{10, 100, 1\,000, 10\,000\}$ and pattern lengths $m = \{8, 16, 32, 64\}$

almost achieves 100 MB/s (the worst ones here, SOG and RK, are 10 times slower). However, the parameters of MAG are calculated using a formula designed for random text, so better results are possible with tuning the parameters for a particular dataset (actually, we achieved almost 200 MB/s for above example). Yet, this approach is rather inelegant and time-consuming in the construction phase.

In Figure 7 the number of patterns r is fixed (1 000), but m grows. We chose three variants, `mag` (the HAM variant) for all datasets, `mag_l4` for `english` and `proteins`, and `mag_dna_opt_l2` for `dna` set. MAG usually wins on `english` and `proteins` (except for the shortest patterns), yet is dominated by a few algorithms on `dna`. Overall, in the experiments the toughest competitor to MAG was SBDM, but in some cases (the shortest patterns ($m = 8$) on `english` and `proteins`) the winner was SOG.

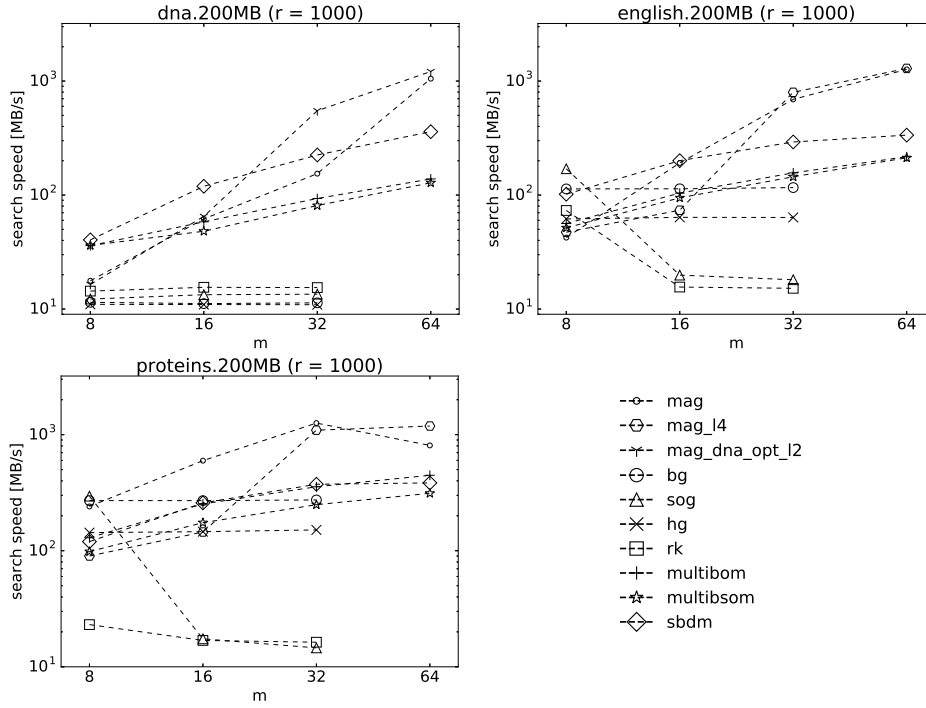

Figure 7. Search speeds for the number of patterns $r = 10\,00$ and varying pattern length m

4.3 MAG on ETDC

It is interesting to test multiple pattern matching also on compressed text. To this end, we chose ETDC (see Section 3.6), a popular byte code scheme, applied to the words of the text. The benefits of using ETDC are less text to search in, possibly less RAM used, and (as we expect) reduced search time. Similarly to the experiments on plain text, we use now different alphabet mapping methods. In this experiment we use $\sigma' = 26$ for `etdc_mag`, and for other variants the same σ' parameters as in the experiments with plain text. The variant naming convention is preserved. The ETDC-encoded `english` text (including the helper array A) takes between 33 (for the smallest A) and 35 (for the largest A) percent of the original text.

In Figure 8 the performance difference between all variants of MAG algorithm adapted for ETDC is presented. The ETDC variant has quite better performance than the corresponding solution on plain text. ETDC compression allows to obtain the compression ratio of factor 2.9 (as the ETDC-encoded file, the dictionary and the array A take in total 35 % of the original file). The most successful variant is `etdc_mag_l4`, which achieves not only the highest speed among all the variants but

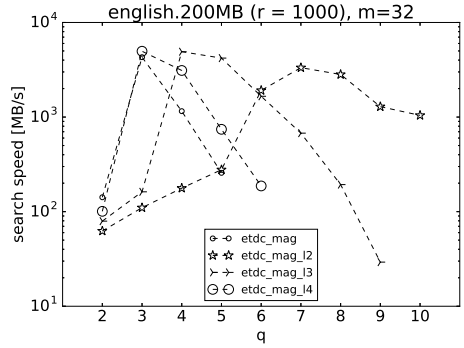

Figure 8. Search speeds for the pattern length $m = 32$ and varying q

then also makes use of a relatively small q (4), beneficial for RAM usage. This figure also shows the growth of the optimal q for decreasing ℓ parameter (i.e., for $\ell = 4$ the optimal q is 3, for $\ell = 3$ the optimal q is 4 and for $\ell = 2$ it is 7).

It is not trivial to compare the results of MAG in our two scenarios: on plain text and ETDC-compressed text. The reason is that in the compressed scenario the patterns must consist of words (which are obviously of varying length) and the length (in bytes) of the ETDC codewords for the input words tends to differ. Taking these into consideration, we decided to compare these algorithms by calculating the average length of decoded patterns (for the ETDC case) and then execute MAG tests with corresponding pattern lengths on plain text.

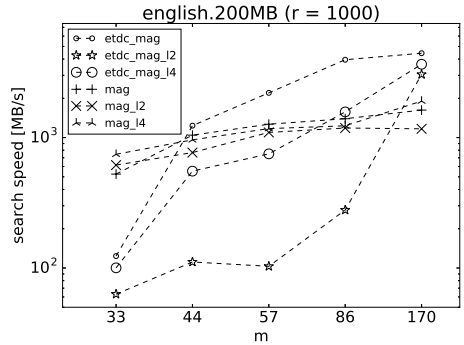

Figure 9. Search speeds (comparison of plain text variants against ETDC variants) for varying pattern lengths $m = \{33, 44, 57, 86, 170\}$

Figure 9 presents search speeds of MAG on plain text and MAG on ETDC-compressed text with varying pattern length in $\{33, 44, 57, 86, 170\}$ (averages for ETDC). The corresponding word counts (in case of ETDC) for each pattern length are $\{6, 8, 10, 16, 32\}$. As expected, the performance of MAG on ETDC data is much

better than on plain text. The speed of the best result of MAG on ETDC is by a factor of 2.8 higher (for $m = 86$ (16)) compared to the corresponding best result of MAG on plain text.

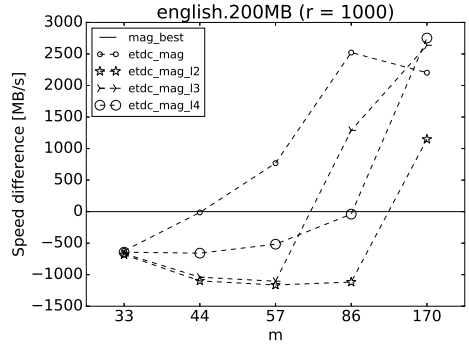

Figure 10. Difference of search speeds between the best plain text variant and ETDC variants for varying pattern lengths $m = \{33, 44, 57, 86, 170\}$

In Figure 10 we compare variants of MAG on ETDC (`etdc_mag`, `etdc_mag_l2`, `etdc_mag_l3`, `etdc_mag_l4`) against the best result of MAG on plain text (`mag`, `mag_l2`, `mag_l3`, `mag_l4`) which is labeled as `mag_best`. Overall, we can see a clear upward trend in the speed growth when pattern length is increasing. Note that the machine word size (64 bits) is limiting the MAG (`mag_best`) performance in terms of large pattern sizes, this is why we expected drastic speed increase for patterns longer than 64 characters. On the other hand, a longer machine word can also improve the performance of ETDC variants for even longer patterns.

5 CONCLUSIONS AND FUTURE WORK

Multiple string matching is one of the most explored problems in stringology. The presented algorithm, MAG, usually wins with its competitors on the three test datasets (`english`, `proteins` and `dna`). We discuss the pros and cons of various alternatives to achieve a possibly best combination of techniques. The main contribution and one of the key successful ideas was alphabet quantization such as bin-packing which is performed in a greedy manner, after sorting the original alphabet by frequency (HAM). We also proposed a different implementation of alphabet quantization, called combined alphabet mapping (CAMq, in some variants), that has a few advantages like faster preprocessing (there is no need to create a histogram), no array access (because there is no histogram) and less operations. The disadvantage, which may be called reduced flexibility, is strong dependence on machine word size, what significantly constraints the values of q and ℓ.

There is a number of interesting questions that we can present here. We analytically showed that the presented approach is sublinear on average, yet not average optimal. Therefore, is it possible to choose the algorithm's parameters in order to reach average optimality (for $m = O(w)$)?

Our experiments confirmed that dense codes (ETDC) for words not only serve for compressing data (texts), but also enable faster searching, for long enough patterns. Thanks to the fact that the encoded pattern is much shorter than the original, the actual input pattern length may be increased, which effectively raises the limit on the machine word size and provides better performance for longer patterns.

Real computers nowadays have a hierarchy of caches in their CPU-related architecture and it could be interesting to apply the I/O model (or cache-obvious model) for the multiple pattern matching problem. The cache efficiency issue may be crucial for very large pattern sets.

The underexplored power of the SIMD instructions also seems to offer great opportunities, especially for bit-parallel algorithms.

REFERENCES

[1] Aho, A. V.—Corasick, M. J.: Efficient String Matching: An Aid to Bibliographic Search. Communications of the ACM, Vol. 18, 1975, No. 6, pp. 333–340, doi: 10.1145/360825.360855.

[2] Allauzen, C.—Raffinot, M.: Factor Oracle of a Set of Words. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.

[3] Baeza-Yates, R. A.—Gonnet, G. H.: A New Approach to Text Searching. Communications of the ACM, Vol. 35, 1992, No. 10, pp. 74–82, doi: 10.1145/135239.135243.

[4] Boyer, R. S.—Moore, J. S.: A Fast String Searching Algorithm. Communications of the ACM, Vol. 20, 1977, No. 10, pp. 762–772, doi: 10.1145/359842.359859.

[5] Brisaboa, N. R.—Fariña, A.—Navarro, G.—Paramá, J. R.: New Adaptive Compressors for Natural Language Text. Software: Practice and Experience, Vol. 38, 2008, No. 13, pp. 1429–1450, doi: 10.1002/spe.882.

[6] Burkhardt, S.—Kärkkäinen, J.: Better Filtering with Gapped q-Grams. Fundamenta Informaticae, Vol. 56, 2003, No. 1–2, pp. 51–70.

[7] Cantone, D.—Faro, S.—Giaquinta, E.: A Compact Representation of Nondeterministic (Suffix) Automata for the Bit-Parallel Approach. Information and Computation, Vol. 213, 2012, pp. 3–12, doi: 10.1016/j.ic.2011.03.006.

[8] Commentz-Walter, B.: A String Matching Algorithm Fast on the Average. In: Maurer, H. A. (Ed.): Automata, Languages and Programming (ICALP 1979). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 71, 1979, pp. 118–132, doi: 10.1007/3-540-09510-1_10.

[9] Crochemore, M.—Czumaj, A.—Gąsieniec, L.—Lecroq, T.—Plandowski, W.—Rytter, W.: Fast Practical Multi-Pattern Matching. Information Processing Letters, Vol. 71, 1999, No. 3–4, pp. 107–113, doi: 10.1016/s0020-0190(99)00092-7.

[10] Crochemore, M.—Hancart, Ch.—Lecroq, T.: Algorithms on Strings. Cambridge University Press, New York, USA, 2007, doi: 10.1017/cbo9780511546853.

[11] Crochemore, M.—Rytter, W.: Text Algorithms. Oxford University Press, 1994.

[12] Dori, S.—Landau, G. M.: Construction of Aho Corasick Automaton in Linear Time for Integer Alphabets. Information Processing Letters, Vol. 98, 2006, No. 2, pp. 66–72, doi: 10.1016/j.ipl.2005.11.019.

[13] Faro, S.—Külekci, M. O.: Fast Multiple String Matching Using Streaming SIMD Extensions Technology. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (Eds.): String Processing and Information Retrieval (SPIRE 2012). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7608, 2012, pp. 217–228, doi: 10.1007/978-3-642-34109-0_23.

[14] Faro, S.—Külekci, M. O.: Towards a Very Fast Multiple String Matching Algorithm for Short Patterns. Proceedings of the Prague Stringology Conference, Prague, 2013, pp. 78–91.

[15] Fisk, M.—Varghese, G.: Fast Content-Based Packet Handling for Intrusion Detection. Technical report, DTIC Document, 2001, doi: 10.21236/ada406413.

[16] Fredriksson, K.: Shift-Or String Matching with Super-Alphabets. Information Processing Letters, Vol. 87, 2003, No. 4, pp. 201–204, doi: 10.1016/s0020-0190(03)00296-5.

[17] Fredriksson, K.: Succinct Backward-DAWG-Matching. ACM Journal of Experimental Algorithmics, Vol. 13, 2009, Art. No. 8, doi: 10.1145/1412228.1455263.

[18] Fredriksson, K.—Grabowski, Sz.: Average-Optimal String Matching. Journal of Discrete Algorithms, Vol. 7, 2009, No. 4, pp. 579–594, doi: 10.1016/j.jda.2008.09.001.

[19] Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.

[20] Horspool, R. N.: Practical Fast Searching in Strings. Software: Practice and Experience, Vol. 10, 1980, No. 6, pp. 501–506, doi: 10.1002/spe.4380100608.

[21] Kandhan, R.—Teletia, N.—Patel, J. M.: SigMatch: Fast and Scalable Multi-Pattern Matching. Proceedings of the VLDB Endowment, Vol. 3, 2010, No. 1–2, pp. 1173–1184, doi: 10.14778/1920841.1920987.

[22] Knuth, D. E.—Morris, J. H.—Pratt, V. R.: Fast Pattern Matching in Strings. SIAM Journal on Computing, Vol. 6, 1977, No. 1, pp. 323–350, doi: 10.1137/0206024.

[23] Kouzinopoulos, C. S.—Michailidis, P. D.—Margaritis, K. G.: Parallel Processing of Multiple Pattern Matching Algorithms for Biological Sequences: Methods and Performance Results. Chapter 8. Systems and Computational Biology – Bioinformatics and Computational Modeling, InTech, 2011, pp. 161–182, doi: 10.5772/18488.

[24] Kouzinopoulos, C. S.—Michailidis, P. D.—Margaritis, K. G.: Multiple String Matching on a GPU Using CUDAs. Scalable Computing: Practice and Experience, Vol. 16, 2015, No. 2, pp. 121–137, doi: 10.12694/scpe.v16i2.1085.

[25] Navarro, G.—Fredriksson, K.: Average Complexity of Exact and Approximate Multiple String Matching. Theoretical Computer Science, Vol. 321, 2004, No. 2–3, pp. 283–290, doi: 10.1016/j.tcs.2004.03.058.

[26] Navarro, G.—Raffinot, M.: Fast and Flexible String Matching by Combining Bit-Parallelism and Suffix Automata. ACM Journal of Experimental Algorithmics, Vol. 5, 2000, Art. No. 4, doi: 10.1145/351827.384246.

[27] Navarro, G.—Raffinot, M.: Flexible Pattern Matching in Strings – Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, 2002. ISBN 0-521-81307-7, 280 pp., doi: 10.1017/cbo9781316135228.

[28] Peltola, H.—Tarhio, J.: Alternative Algorithms for Bit-Parallel String Matching. In: Nascimento, M. A., de Moura, E. S., Oliveira, A. L. (Eds.): String Processing and Information Retrieval (SPIRE 2003). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2857, 2003, pp. 80–93, doi: 10.1007/978-3-540-39984-1_7.

[29] Rivals, E.—Salmela, L.—Tarhio, J.: Exact Search Algorithms for Biological Sequences. Chapter 5. Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications, John Wiley and Sons, Inc., 2010, pp. 91–111, doi: 10.1002/9780470892107.ch5.

[30] Salmela, L.—Tarhio, J.—Kytöjoki, J.: Multipattern String Matching with q-Grams. ACM Journal of Experimental Algorithmics, Vol. 11, 2006, Art. No. 1.1, doi: 10.1145/1187436.1187438.

[31] Susik, R.—Grabowski, Sz.—Fredriksson, K.: Multiple Pattern Matching Revisited. Proceedings of the Prague Stringology Conference, Prague, 2014, pp. 59–70.

[32] Wu, S.—Manber, U.: A Fast Algorithm for Multi-Pattern Searching. Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[33] Yang, P.—Liu, L.—Fan, H.—Huang, Q.: Fast Multi-Pattern String Matching Algorithms Based on q-Grams Bit-Parallelism Filter and Hash. In: Lu, W., Cai, G., Liu, W., Xing, W. (Eds.): Proceedings of the 2012 International Conference on Information Technology and Software Engineering. Springer, Berlin, Heidelberg, Lecture Notes in Electrical Engineering, Vol. 211, 2013, pp. 487–495, doi: 10.1007/978-3-642-34522-7_52.

[34] Yao, A. C.-C.: The Complexity of Pattern Matching for a Random String. SIAM Journal on Computing, Vol. 8, 1979, No. 3, pp. 368–387, doi: 10.1137/0208029.

Robert Susik received his M.Eng. and Ph.D. degrees from the Lodz University of Technology in 2012 and 2018, respectively. Currently, apart from string matching algorithms, his interests mostly focus on data archiving, machine learning and data science.

Szymon Grabowski received his M.Sc. degree from the University of Lodz in 1996, his Ph.D. degree from AGH University of Science and Technology in Cracow in 2003, and the habilitation degree from the Systems Research Institute of the Polish Academy of Sciences in Warsaw in 2011. His former research, including Ph.D. dissertation, involved nearest neighbor classification methods in pattern recognition, also with applications in image processing. Currently, his main interests are focused on string matching and text indexing algorithms, and data compression. Some of his particular research topics include various approximate string matching problems, compressed text indexes, and XML compression. He has published over 120 papers in journals and conferences. He is currently Professor at the Institute of Applied Computer Science of the Lodz University of Technology.

Kimmo Fredriksson received his computer science M.Sc. and Ph.D. degrees from the University of Helsinki in 1997 and 2001, respectively. His research interests include wide variety of string matching problems as well as indexing techniques for searching in metric spaces. He has published about 60 papers on these topics in international conferences and journals. He has the position of Adjunct Professor at the University of Eastern Finland.