Multiple Pattern Matching Revisited

Robert Susik[1], Szymon Grabowski[1], and Kimmo Fredriksson[2]

[1] Lodz University of Technology, Institute of Applied Computer Science
Al. Politechniki 11, 90–924 Łódź, Poland
{rsusik|sgrabow}@kis.p.lodz.pl
[2] School of Computing, University of Eastern Finland
P.O.B. 1627, FI-70211 Kuopio, Finland
kimmo.fredriksson@uef.fi

Abstract. We consider the classical exact multiple string matching problem. Our solution is based on q-grams combined with pattern superimposition, bit-parallelism and alphabet size reduction. We discuss the pros and cons of the various alternatives of how to achieve best combination. Our method is closely related to previous work by (Salmela et al., 2006). The experimental results show that our method performs well on different alphabet sizes and that they scale to large pattern sets.

Keywords: combinatorial problems, string algorithms, q-grams, word-level parallelism

1 Introduction

Multiple pattern matching is a classic problem, with about 40 years of history, with applications in intrusion detection, anti-virus software and bioinformatics, to name a few. The problem can be stated as follows: Given text T of length n and pattern set $\mathcal{P} = \{P_1, \ldots, P_r\}$, 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. The multiple pattern matching problem is a straightforward generalization of single pattern matching and it is no surprise that many techniques worked out for a single pattern are borrowed in efficient algorithms for multiple patterns.

1.1 Related work

The classical algorithms for the present problem can be roughly divided into three different categories, (*i*) prefix searching, (*ii*) suffix searching and (*iii*) factor searching. Another way to classify the solutions is to say that 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 we refer the reader e.g. to [21,16,9]. We briefly review some of them in the following.

Perhaps the most famous solution to the multiple pattern matching problem is the Aho–Corasick (AC) [1] algorithm, which works in linear time (prefix-based approach). It builds a pattern trie with extra (failure) links and actually generalizes the Knuth–Morris–Pratt algorithm [18] for a single pattern. More precisely, AC total time is $O(M + n + z)$, where M, the sum of pattern lengths, is the preprocessing cost, and z is the total number of pattern occurrences in T. Recently Fredriksson and

Robert Susik, Szymon Grabowski, Kimmo Fredriksson: Multiple Pattern Matching Revisited, pp. 59–70.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

Grabowski [15] 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 the same ideas as the current work.

Another classic algorithm is Commentz–Walter [7], 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 [12,21] may be considered its more practical simplification, exactly in the way that Boyer–Moore–Horspool (BMH) [17] 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 [24]. These methods are based on backward matching over a sliding text window, which is shifted based on some rule, and the hope is that many text characters can be skipped altogether.

The first factor based algorithms were DAWG-match [8] and MultiBDM [10]. 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. 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) [14], 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 [19] complexity $O(n \log_\sigma(rm)/m)$.

Bit-parallelism can be used to replace the various automata in the previous methods to obtain very simple and very efficient variants of many classical algorithms. The classic method for a single pattern is Shift-Or [3]. The idea is to encode (nondeterministic) 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 [22,6]. 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.

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 [11] 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 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. Sec-

tion 4 contains (preliminary) experimental results. The last section concludes and points some avenues for pursuing further research.

2 On q-grams and superimposition

A q-gram is (usually) a contiguous substring (factor) of q characters of a string, although non-contiguous q-grams have been considered [5]. 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 interesting in our point of view. 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, see [13].

In our case q-grams are interesting as combined with a technique called superimposition. Consider a set of patterns $\mathcal{P} = \{P_1, \ldots, P_r\}$. 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 $\mathcal{P}$. That is, if $T[i+j] \in P[j]$, for all $j \in 1, \ldots, m$, then $T[i \ldots i+m-1]$ may match with some pattern in $\mathcal{P}$.

For example, if $\mathcal{P} = \{abba, bbac\}$, the superimposed pattern will be $P = \{a,b\}\{b\}\{a,b\}\{a,c\}$, 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 mean 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 $j \in 1, \ldots, m-q+1$, 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 $j \in 1, \ldots, \lfloor m/q \rfloor$, 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 qth 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 $\hat{P}_i = \{P_i[1 \ldots m], P_i[2 \ldots m], \ldots, P_i[q-1 \ldots m]\}$, and then generate $\hat{P}'_i$, and finally superimpose them.

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

- pattern length is large $(m - q + 1)$, which means less verifications
- text length is practically unaffected $(n - q + 1)$

Non-overlapping:

- 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. [23] presented three algorithms combining the known mechanisms: Shift-Or, BNDM [20] and BMH, with overlapping q-grams; the former two of these algorithms are bit-parallel ones. The resulting algorithm 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 [25] tries to solve the problem by combining BG with hashing.

In general, 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 [15].

3 Our algorithm

In [15] 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:

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

In other words, k different alignments of the original pattern P is generated, each alignment containing only every kth 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

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

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 kth character of T. Fig. 1 serves as an illustration.

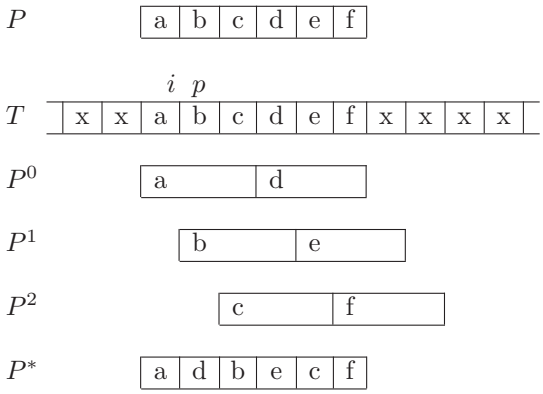

Figure 1. An example. Assume that $P =$ `abcdef` occurs at text position $T[i \ldots i+m-1]$, and that $k = 3$. The current text position is $p = 10$, and $T[p] =$ `b`. The next character the algorithm reads is $T[p+k] = T[13] =$ `e`. This triggers a match of $P^{p \mod k} = P^1$, and the text area $T[p-1 \ldots p-1+m-1] = T[i \ldots i+m-1]$ 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 kth 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 at most $O(n/k)$ 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 [26].

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

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

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 actually search the superimposed pattern, we use FAOSO [15], 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 [15]. 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. [23] have tried a similar approach, but dismissed it early because it did not look promising for short patterns in their tests.

Implementation. In the algorithms' point of view 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 $\sum_{i=1}^{q} S[i] \cdot \sigma^{i-1}$ for a q-gram $S[1 \ldots q]$. 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 length q strings as super characters, we may 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.

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.

Alphabet mapping on the *q*-grams. We note that the above method can be applied also on the q-gram alphabet. This allows a precise control of the table size, and combined with hashing, it can accommodate very large q as well. That is, we want to

1. Choose some (possibly very large) q;
2. compute the q-gram frequencies on the pattern set (using e.g. hashing to avoid possibly large tables);
3. choose some suitable σ', the size of the mapped q-gram alphabet;
4. use method of choice (e.g. bin-packing) to reduce the number of q-grams, i.e. map the q-grams to range $0, \ldots, \sigma' - 1$;
5. use hashing to store the mapping, along with the corresponding bitvectors needed by FAOSO.

Combined alphabet mapping and *q*-gram generation. 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 $S[1 \ldots q]$ 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

$$s = (S[1] \,\&\, b) + (S[2] \,\&\, b) \ll \ell + (S[3] \,\&\, b) \ll 2\ell + \cdots + (S[q] \,\&\, b) \ll (q-1)\ell,$$

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

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 do

$$s = (((x \gg 1) \,\&\, \texttt{0x03030303}) * \texttt{0x40100401}) \gg 24,$$

where x is the 32-bit integer containing the 4 chars $S[1 \dots 4]$. Assuming 4 letter DNA alphabet, the right shift (by 1) and the (parallel) masking generate 2-bit unique (and case insensitive) codes for all 4 characters. If the alphabet is larger (some DNA sequences have rare additional symbols), those will be mapped in the same range, $0, \dots, 3$. The multiplication then shifts and adds all those codes into an 8-bit quantity, and the final shift moves the 4-gram to the low order bits. Larger q-grams can be obtained by repeating the code.

We leave the implementation to future work.

4 Experimental results

In order to evaluate the performance of our approach, we run a few experiments, using the 200 MB versions of selected datasets (`dna`, `english` and `proteins`) from the widely used Pizza & Chili corpus (`http://pizzachili.dcc.uchile.cl/`).

We test the following algorithms:

- BNDM on q-grams (BG) [23],
- Shift-Or on q-grams (SOG) [23],
- BMH on q-grams (HG) [23],
- Rabin-Karp algorithm combined with binary search and two-level hashing (RK) [23],
- Multibom and Multibsom are variants of the Set Backward Oracle Matching algorithm [2],
- Succinct Backward DAWG Matching algorithm (SBDM) [14],
- Multi AOSO on q-Grams (MAG) (this work).

All codes were obtained from the original authors. Our MAG was implemented in C++ and compiled with `g++` version 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. The test machine was equiped with 4 GB of 1333 MHz DDR3 RAM and running Ubuntu 64-bit OS with kernel 3.11.0-17.

In Fig. 2 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 ($m = 64$). Our algorithm, MAG, depends on several parameters: k, q and U. The first two were explained earlier, and U serves for an unrolling technique which reduces the number of executed conditionals in the search code (for more details, see [15]). We use two settings for MAG. In one of them we chose the best configurations of k, q and U, for each dataset and each value of r and m separately; this variant is presented as MAG-tuned. It dominates for longer patterns (32, 64) and its performance is mixed for $m = 8$ and $m = 16$. As expected, for all algorithms the search speed deteriorates with the number of patterns, and for $r = 10,000$ and relatively long patterns ($m = 32$) only MAG slightly exceeds 100 MB/s (the worst ones here, SOG and RK, are 10 times slower).

Although the "optimal" MAG settings may be found in the construction phase, assuming the patterns are randomly taken from the text, this approach is rather inelegant (and the tuning phase may be time-consuming). Therefore, we ran another test in which the parameters U and k (yet not q) are set for a particular dataset, m and r according to the following simple rules found experimentally: if the "best" value of q is greater than 5, we set $U = 8$ and $k = 1$, otherwise we set $U = 4$ and $k = 2$. The case of `english` and $m = 8$ is an exception, where $U = 8$ and $k = 1$ was

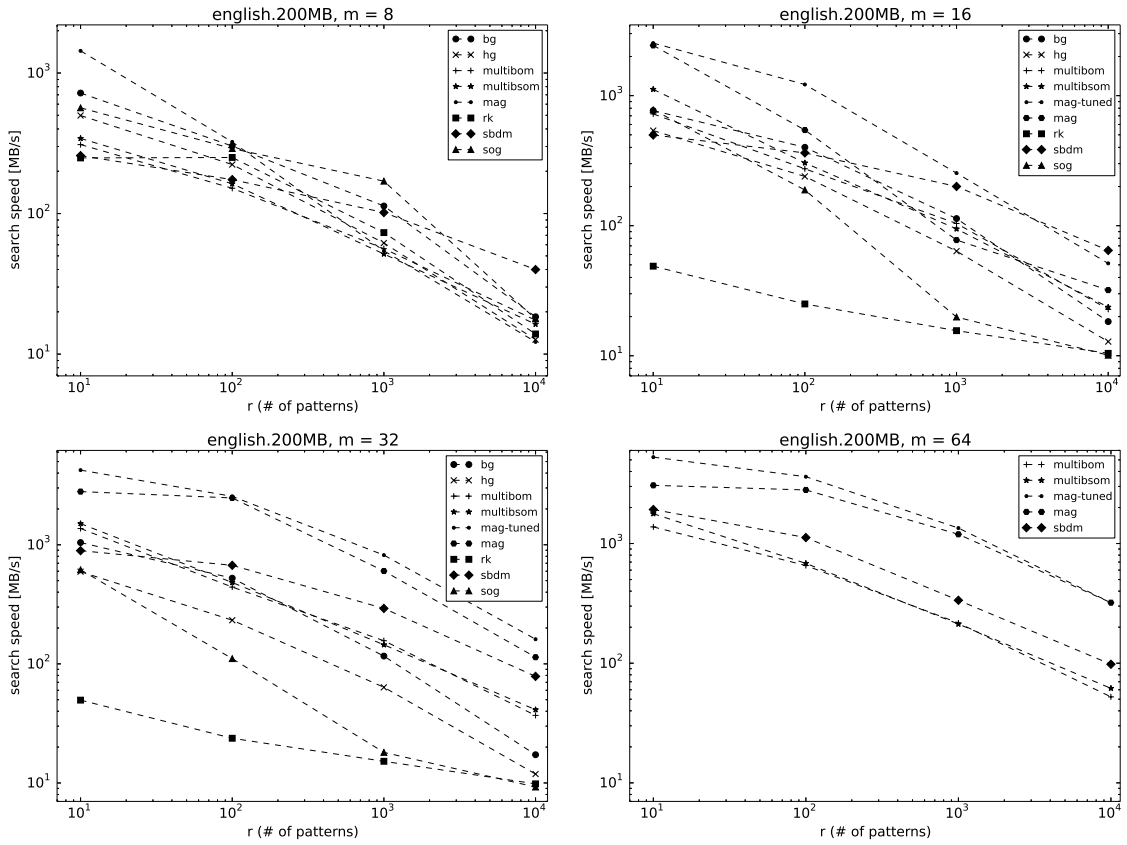

Figure 2. `english`, search speeds (MB/s) for varying number of patterns r. MAG is the same as MAG-tuned for $m = 8$, hence the "mag-tuned" points for this case are not presented.

set no matter the value of q. These results are presented on the plots as MAG. As expected, MAG is slower than MAG-tuned, but the differences are not huge.

In Fig. 3 the number of patterns r is fixed (1000), but m grows. 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 winner was SOG.

We also show how MAG performance changes with growing q (Fig. 4). As expected, larger q makes sense for large r, but a too large value of it slows down the search, presumably to many cache misses. The used MAG variant is MAG-tuned, the alphabet is quantized for all datasets. The new alphabet size, σ', was found (separately for each case) from the set $\{4, 5, 13, 14, 22\}$. Note that due to the quantization the original alphabet size does not (significantly) affect the choice of q.

5 Conclusions and future work

Multiple string matching is one of the most exploited problems in stringology. It is hard to find really novel ideas for this idea, and our work can also be seen as a new and quite successful combination of known building bricks. The presented algorithm, MAG, usually wins with its competitors on the three test datasets (`english` and `proteins`, `dna`). One of the key successful ideas was alphabet quantization (binning), which is performed in a greedy manner, after sorting the original alphabet by frequency. In the future, we are going to try other quantization techniques, also for

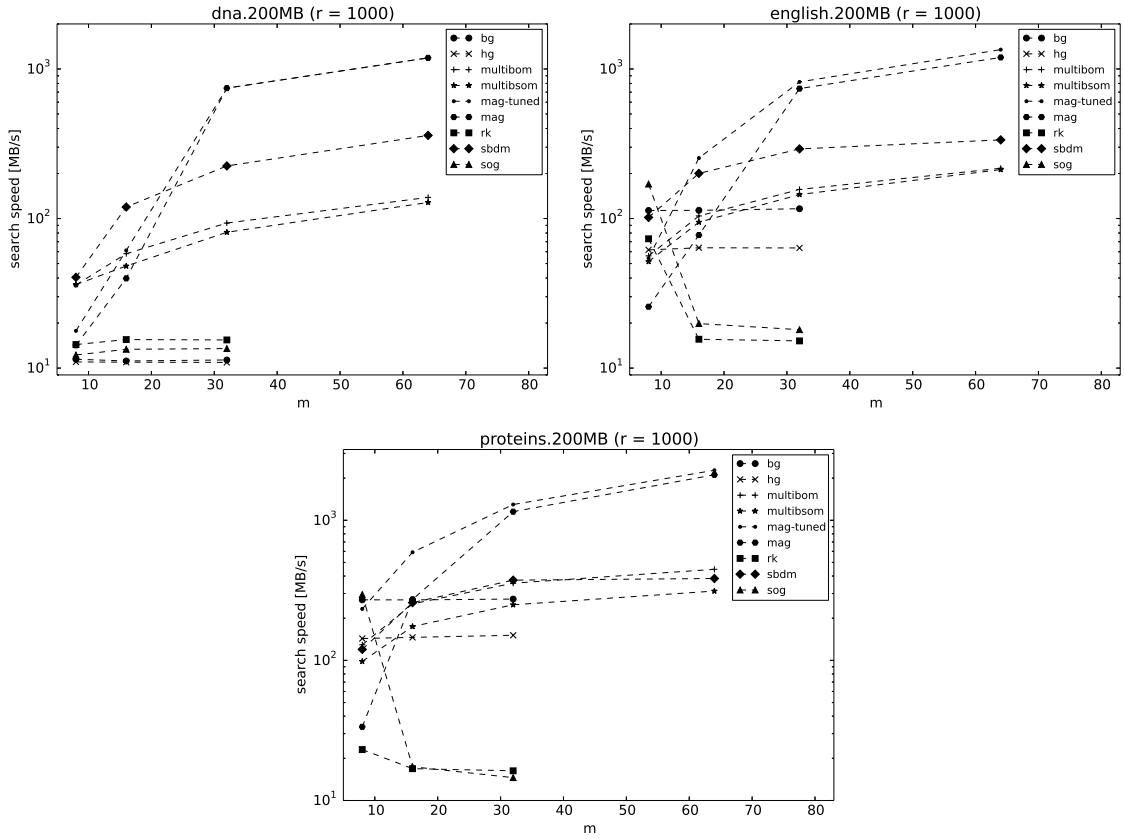

Figure 3. Search speeds for the number of patterns $r = 1000$ and varying pattern length m.

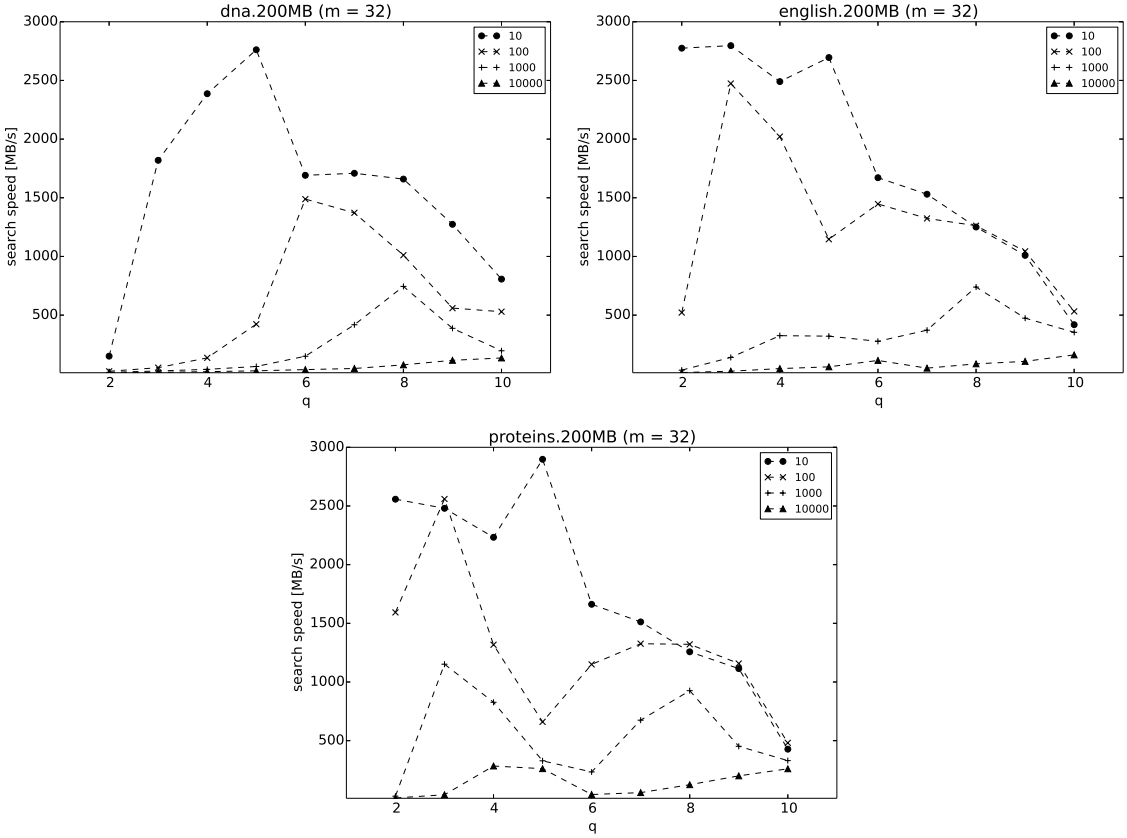

Figure 4. Search speeds for the pattern length $m = 32$ and varying q.

quantization of the alphabet built on q-grams. This could give further improvement in the algorithm performance and savings in memory consumption.

Apart from the mentioned issue, there are a number of interesting questions that we can pose 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 $m = O(w)$)?

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.

It was reported that dense codes (e.g., ETDC) for words or q-grams not only serve for compressing data (texts), but also enable faster pattern searches. Multiple pattern searching over such compressed data seems unexplored yet and it is interesting to apply our algorithm for this scenario (our preliminary results are rather promising).

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18(6) 1975, pp. 333–340.
2. C. Allauzen and M. Raffinot: *Factor oracle of a set of words*, Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.
3. R. A. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, pp. 74–82.
4. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
5. S. Burkhardt and J. Kärkkäinen: *Better filtering with gapped q-grams.* Fundam. Inform., 56(1-2) 2003, pp. 51–70.
6. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach.* Inf. Comput., 213 2012, pp. 3–12.
7. B. Commentz-Walter: *A string matching algorithm fast on the average*, in Proceedings of the 6th International Colloquium on Automata, Languages and Programming, H. A. Maurer, ed., no. 71 in Lecture Notes in Computer Science, Graz, Austria, 1979, Springer, pp. 118–132.
8. M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter: *Fast practical multi-pattern matching.* Inf. Process. Lett., 71(3-4) 1999, pp. 107–113.
9. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, New York, USA, 2007.
10. M. Crochemore and W. Rytter: *Text algorithms*, Oxford University Press, 1994.
11. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming simd extensions technology*, in SPIRE, L. Calderón-Benavides, C. N. González-Caro, E. Chávez, and N. Ziviani, eds., vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 217–228.
12. M. Fisk and G. Varghese: *Fast content-based packet handling for intrusion detection*, tech. rep., DTIC Document, 2001.
13. K. Fredriksson: *Shift–or string matching with super-alphabets.* Information Processing Letters, 87(1) 2003, pp. 201–204.
14. K. Fredriksson: *Succinct backward-DAWG-matching.* J. Exp. Algorithmics, 13 2009, pp. 1.8–1.26.
15. K. Fredriksson and S. Grabowski: *Average-optimal string matching.* J. Discrete Algorithms, 7(4) 2009, pp. 579–594.
16. D. Gusfield: *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
17. R. N. Horspool: *Practical fast searching in strings.* Softw., Pract. Exper., 10(6) 1980, pp. 501–506.

18. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(1) 1977, pp. 323–350.
19. G. Navarro and K. Fredriksson: *Average complexity of exact and approximate multiple string matching.* Theoretical Computer Science A, 321(2–3) 2004, pp. 283–290.
20. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata.* ACM Journal of Experimental Algorithmics (JEA), 5 2000, p. article 4, 36 pages. `http://www.jea.acm.org/2000/NavarroString`.
21. G. Navarro and M. Raffinot: *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 pages.
22. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE2003), LNCS 2857, Springer–Verlag, 2003, pp. 80–94.
23. L. Salmela, J. Tarhio, and J. Kytöjoki: *Multipattern string matching with q-grams.* ACM Journal of Experimental Algorithmics, 11 2006.
24. S. Wu and U. Manber: *A fast algorithm for multi-pattern searching*, Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.
25. P. Yang, L. Liu, H. Fan, and Q. Huang: *Fast multi-pattern string matching algorithms based on q-grams bit-parallelism filter and hash*, in Proceedings of the 2012 International Conference on Information Technology and Software Engineering, vol. 211 of LNEE, Springer, 2013, pp. 487–495.
26. A. C. Yao: *The complexity of pattern matching for a random string.* SIAM Journal on Computing, 8(3) 1979, pp. 368–387.