Fast Indexes for Gapped Pattern Matching

Manuel Cáceres[1], Simon J. Puglisi[2], and Bella Zhukova[2](✉)

[1] Department of Computer Science, University of Chile, Santiago, Chile
mcaceres@dcc.uchile.cl

[2] Department of Computer Science, University of Helsinki, Helsinki Institute for Information Technology (HIIT), Helsinki, Finland
{puglisi,bzhukova}@cs.helsinki.fi

Abstract. We describe indexes for searching large data sets for variable-length-gapped (VLG) patterns. VLG patterns are composed of two or more subpatterns, between each adjacent pair of which is a gap-constraint specifying upper and lower bounds on the distance allowed between subpatterns. VLG patterns have numerous applications in computational biology (motif search), information retrieval (e.g., for language models, snippet generation, machine translation) and capture a useful subclass of the regular expressions commonly used in practice for searching source code. Our best approach provides search speeds several times faster than prior art across a broad range of patterns and texts.

1 Introduction

In the classic pattern matching problem, we are given a string P (the pattern or query) and asked to report all the positions where it occures in another (longer) string T (the text). This problem has been very heavily studied and has applications throughout computer science.

In this paper we consider a variant on the classic pattern matching problem, called variable length gap (VLG) pattern matching. In VLG matching, the query P is not a single string but is composed of $k \geq 2$ strings (subpatterns) that must occur in order in the text. Between each subpattern, a number of characters may be allowed to occur, an upper and lower bound on which is specified as part of the query. Formally, our problem is as follows.

Definition 1 (Variable Length Gap (VLG) Pattern Matching [4]). *Let* T *be a string of* n *symbols drawn from alphabet* Σ *and* P *be a pattern consisting of* $k \geq 2$ *subpatterns (i.e. strings)* $p_0, \ldots, p_{k-1}$*, each consisting of symbols also drawn from* Σ*, and having lengths* $m_0, \ldots, m_{k-1}$*, and* $k-1$ *gap constraints* $C_0, \ldots, C_{k-2}$*, such that* $C_i = \langle \delta_i, \Delta_i \rangle$ *with* $0 \leq \delta_i \leq \Delta_i < n$ *specifies the smallest* (δ_i) *and largest* (Δ_i) *allowable distance between a match of* p_i *and* p_{i+1} *in* T*. Find all matches—reported as* k*-tuples* $i_0, \ldots, i_{k-1}$ *where* i_j *is the starting position for subpattern* p_j *in* T*—such that all gap constraints are satisfied.*

This research is supported by Academy of Finland through grant 319454.

In computational biology, VLG matching is used in the discovery of and search for *motifs*— i.e. conserved features—in sets of DNA and protein sequences (see, e.g., [18,20]). For example, the following is a protein motif from the rice genome (see [18]) expressed as a VLG pattern:

$$\texttt{MT}[115,\ 136]\texttt{MTNTAYGG}[121,\ 151]\texttt{GTNGAYGAY}.$$

A similar motif concept in music information retrieval means VLG matching also finds applications in mining and searching for characteristic melodies [7] and other musical structures [8,9] in sequences of musical notes expressed in chromatic or diatonic notation. Bader et al. [1] point out several more applications of VLG matching in information retrieval and related fields such as natural language processing (NLP) and machine translation. For example, Metzler and Croft [17] define a language model in which query terms occuring within as certain window of each other must be found (in NLP, such terms are said to be *colocated*). Locating tight windows of a document containing the set of words contained in a search engine query is the problem of query-biased snippet generation [22]. In machine translation, VLG matching is used to derive rule sets from text collections to boost effectiveness of automated translation systems [15].

VLG matching has a big parameter space and it is easy to think of pathological combinations of pattern and text that lead to an exponential number of matches. Fortunately, however, in practice the problem gets naturally constrained in important ways. The gap constraints are always bound the length of documents under consideration, which in the case of source code or web pages means that usually δ_i and Δ_i (and so their difference) are in (at most) the tens-of-kilobytes range. In genomics and proteomics maximum gaps tend to be around 100 characters or so (see, e.g., [18,20]).

Because of the interesting and useful applications outlined above, VLG matching has received a great deal of attention in the past 20 years. The vast majority of previous work deals with the *online* version of the problem in which both the pattern P and the text T are previously unseen and cannot be preprocessed [2,4,5,9,12,18,21]. Our concern in this paper is the *offline* version of the problem, where T is known in advance and can be preprocessed and an index structure built and stored to later support fast search for previously unseen VLG patterns (the stream of which is assumed to be large, practically infinite). Almost all work on the offline problem is of theoretical interest [3,14]. The exception is the recent work of Bader, Gog, and Petri [1], who develop methods for the *offline* setting that use a combination of suffix arrays [16] and wavelet trees [11,19]. Bader et al. show that their index is an order of magnitude faster at VLG matching than are online methods, and several times faster than q-gram-based indexes, the likes of which were behind Google Code Search [6].

Contribution. Our main contribution in the paper is to show that in practice, on a broad range of inputs typical in real applications of VLG matching, simple algorithms based on intersecting ranges of the suffix array corresponding to subpattern occurrences can be made very fast in practice, and comfortably outperform state-of-the-art methods based on wavelet trees.

We emphasise that none of our new approaches are particularly exotic. They are, however, very fast, and so represent non-trivial baselines by which future (possibly more exotic) indexes for VLG pattern matching and related problems (such as regex matching) can be meaningfully measured.

Roadmap. The remainder of this paper is as follows. Section 2 then looks at a simple method for solving VLG matching that works by sorting and intersecting ranges of the suffix array that contain the occurrences of subpatterns of the VLG pattern. Sections 3 and 4 evolve this basic idea, presenting the results of small illustrative experiments along the way. In Sect. 5 we compare our best performing method to the recent wavelet-tree-based approach of Bader et al., which represents the current state-of-the-art for indexed pattern matching (details of our test machine and data sets can also be found in Sect. 5). Reflections and directions for future work are then offered in Sect. 6.

2 VLG Matching via Sorting and Scanning Suffix Array Intervals

Essential to the methods for VLG matching we will consider in this and later sections is the *suffix array* [16] data structure. The suffix array of T, $|\mathsf{T}| = n$, denoted SA, is an array $\mathrm{SA}[0..n-1]$, which contains a permutation of the integers $0..n$ such that $\mathsf{T}[\mathsf{SA}[0]..n-1] < \mathsf{T}[\mathsf{SA}[1]..n-1] < \cdots < \mathsf{T}[\mathsf{SA}[n]..n-1]$. In other words, $\mathsf{SA}[j] = i$ iff $\mathsf{T}[i..n]$ is the j^{th} suffix of T in ascending lexicographical order. Because of the lexicographic ordering, all the suffixes starting with a given substring p of T form an interval $\mathsf{SA}[s..e]$, which can be determined by binary search in $O(|p| \log n)$ time. Clearly the integers in $\mathsf{SA}[s..e]$ correspond precisely to the distinct positions of occurrence of p in T and once s and e are located it is straightforward to enumerate them in time $O(e-s)$.

The starting point for our approaches is a baseline algorithm from the study by Bader et al. called SA-SCAN, which makes use of the suffix array of T. A pseudo-C++ fragment adapted from Bader et al.'s codebase capturing the main thrust of SA-SCAN is shown in Fig. 1. For ease of reading the code here assumes two subpatterns, but is easy to generalize for $k > 2$.

The operation of SA-SCAN can be summarized as follows. First, search for each of the k subpatterns using SA to arrive at k ranges of the SA containing the subpattern occurrences (in the code listing this is acheived by the two `search` method calls). Next, for each range, allocate a memory buffer equal to the range's size and copy the contents of the range from SA to the newly allocated memory and sort the contents of the buffer (positions of subpattern occurrence) into ascending order. Finally, intersect the positions for subpatterns p_0 and p_1 with respect to the gap constraints. Experimenting with SA-SCAN we observed the time taken to find the ranges of subpattern occurrences in SA constituted less than 1% of the overall runtime, with the vast majority of time spent sorting.

```
SA-scan(string_type p1, string_type p2, int min_gap, int max_gap){
    //1:find intervals of SA containing subpattern occurrences
    std::pair<int,int> interval1 = search(p1,T,SA);
    std::pair<int,int> interval2 = search(p2,T,SA);
    //2: copy positions of subpattern occurrence from SA and sort
    int m1 = interval1.second-interval1.first+1;
    int m2 = interval2.second-interval2.first+1;
    int *A = new int[m1];
    int *B = new int[m2];
    std::memcpy(A,SA+interval1.first,m1);
    std::memcpy(B,SA+interval2.first,m2);
    std::sort(A,A+m1);
    std::sort(B,B+m2);
    //3: intersect according to gap constraints
    for(int i=0,j=0; i<m1 && j<m2; i++){
        while(B[j] < (A[i] + min_gap) && j < m2) j++;
        while(j < m2 && B[j] <= (A[i] + max_gap)){
            result.push_back(B[j]);
            j++;
        }
    }
}
```

Fig. 1. A basic C++ implementation of the SA-SCAN VLG matching algorithm suitable for $k = 2$ subpatterns.

Bader et al. use SA-SCAN as a baseline from which to measure the success of their wavelet-tree-based method. SA-SCAN is natural enough, to be sure, but it does look suspiciously like a straw man. To start with, is `std::sort` really the best we can do for sorting those arrays of integers? We replaced the `std::sort` call with a call to an LSD radix sort of our own implementation (using a radix of 256) and replicated an experiment from Bader et al.'s paper, searching several text collections (including web data, source code, DNA, and proteins—see Sect. 5 for more details) for 20 VLG patterns ($k = 2$, $\delta_i, \Delta_i = \langle 100, 110 \rangle$), composed of very frequent subpatterns drawn from the 200 most common substrings of length 3 in each data set.

Figure 2 shows the results obtained on our test machine (see Sect. 5 for specifications). Using radix sort instead of `std::sort`, SA-SCAN becomes at least two times faster on the Kernel and Proteins datasets, almost twice as fast on CC, and more than 30% faster on Para. A large if algorithmically-somewhat-unexciting leap forward—but further improvements are possible[1].

[1] It is possible that further improvements from sorting alone are possible, using a more heavily engineered sort function that our hand-rolled LSD radix sort. Our point here is that sorting is an important dimension along which SA-SCAN can be optimized.

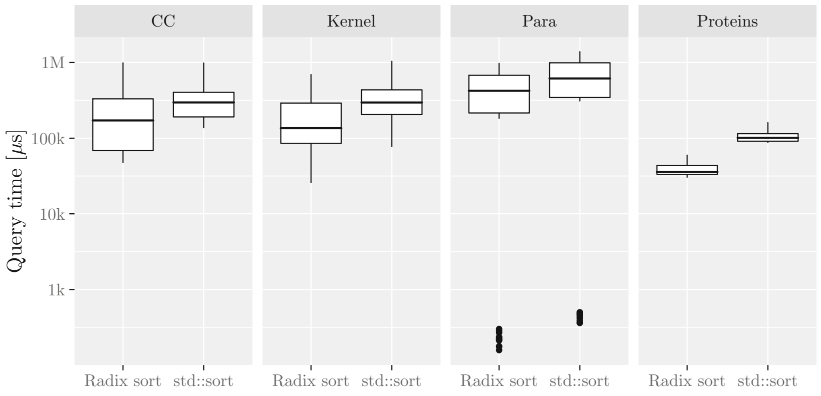

Fig. 2. Time to search a 2GiB subset of the Common Crawl web collection (commoncrawl.org). for 20 VLG patterns ($k = 2$, $\delta_i, \Delta_i = \langle 100, 110 \rangle$), composed of very frequent subpatterns drawn from the 200 most common substrings of length 3 in the collection.

3 Filter, Filter, Sort, Scan

Our first serious embellishment to SA-SCAN aims to avoid sorting the full set of subpattern occurrences by filtering out some of the candidate positions that cannot possibly lead to matches. Specifically, we allocate a bitvector F of n/b bits initially all set to 0. We refer to b as the *block size* of the filter. Logically, each bit represents a block of b contiguous positions in the input text, with the ith bit corresponding to the positions $ib..i(b+1)-1$. In describing the use of the filter we assume two subpatterns p_1 and p_2 (with occurrences in $\mathsf{SA}[s_1..e_1]$ and $\mathsf{SA}[s_2..e_2]$, respectively), but the technique is easy to generalize for $k > 2$.

Having allocated F, we scan the interval $\mathsf{SA}[s_1..e_1]$ containing the occurrences of subpattern p_1 and for each element $i = SA[j]$ encountered, we set bits $F[(i+\delta)/b..(i+\Delta)/b]$ to 1 to indicate that an occurrence of p_2 in any of the corresponding blocks of the input is a potential match. During the scan we also copy elements of the interval to an array A_1 of size $m_1 = e_1 - s_1 + 1$. We then scan the interval $\mathsf{SA}[s_2..e_2]$ containing the occurrences of the second subpattern and for each position i encountered we check $F[i/b]$. If $F[i/b] = 0$ then i cannot possibly be part of a match and can be discarded. Otherwise ($F[i/b] = 1$) we add i to a vector A_2 of candidates. We then sort A_1 and A_2 and intersect them with respect to the gap constraints, the same as in the original SA-SCAN algorithm. The hope is that $|A_2|$ is much less than $e_2 - s_2 + 1$, and so the time spent sorting prior to intersection will be reduced.

There are two straightforward refinements to this approach. The first is to make the initial scan not necessarily over $\mathsf{SA}[s_1..e_1]$, but instead over the smaller of intervals $\mathsf{SA}[s_1..e_1]$ and $\mathsf{SA}[s_2..e_2]$. The only difference is that if the interval for p_2 (the second subpattern) happens to be smaller (i.e. p_2 has less occurrences in T than p_1) then we set bits $F[(i-\delta)/b..(i-\Delta)/b]$ (rather than $F[(i+\delta)/b..(i+\Delta)/b]$)

to 1. Assuming p_1 is in fact more frequent than p_2, the second refinement is to perform a second round of filtering using the contents of A_2. More precisely, having obtained A_2, we clear F (setting all bits to 0) and scan A_2 settings bits $F[(i-\delta)/b..(i-\Delta)/b]$ to 1 for each $i \in A_2$. We then scan A_1 and discard any element i for which $F[i/b]$ now equals 0. Obviously it only makes sense to employ this heuristic if the initial filtering reduced the number of candidates, $|A_2|$, of the second subpattern significantly below m_1. In practice we found $m_2 < m_1/2$ led to a consistent speedup.

Of course, these techniques generalize easily to $k > 2$ subpatterns. The idea is that the output of the intersection of the first two subpatterns then becomes an input interval to be intersected with the third subpattern, and so on.

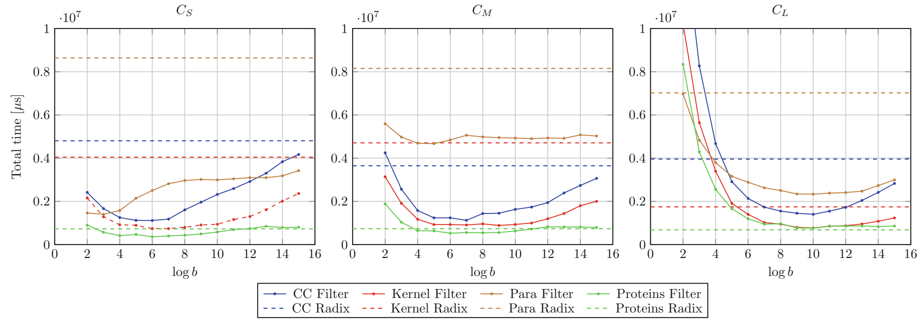

Fig. 3. Effect of filter granularity on search time. Ordinate is runtime in microseconds and mantissa is the logarithm (base 2) of the filter block size. Dashed lines show the time required by SA-SCAN (using radix sort) without any filter. Each plot corresponds to a different set of 20 synthetically generated VLG patterns. All patterns contain 2 of the 200 most frequent subpatterns in each data set. We fixed the gap constraints $C_i = \langle \delta_i, \Delta_i \rangle$ between subpatterns to small ($C_S = \langle 100, 110 \rangle$), medium ($C_M = \langle 1000, 1100 \rangle$), or large ($C_L = \langle 10000, 11000 \rangle$). Section 5 gives more details of data sets and pattern sets.

As Fig. 3 shows, employing F can reduce runtime immensely, but the improvement varies greatly with b. A good choice for b depends on a number of factors. For each occurrence of p_1, we set $\lceil \frac{\Delta - \delta}{b} \rceil$ bits in F. Accesses to F while setting these bits are essentially random (determined by the order of the positions of p_1, which are the lexicographic order of the corresponding suffixes of T), and so it helps greatly if b is chosen so that F, which has size n/b bits, fits in cache. This can be seen in Fig. 3, particularly clearly for the CC, Kernel, and Protein data sets, where performance improves sharply with increasing b until F fits in cache (30 MiB on our test machine) where it quickly stablizes (at $\log b = 3$ for CC and Kernel, and $\log b = 2$ for Protein). Runtimes then remain relatively fast and stable until b becomes so large that the filter lacks specificity, from which point performance gradually degrades. Para has the same trend, though it is not immediately obvious—because the data set is smaller (409MB)

F already fits in L3 cache when $b = 2$. Section 5 gives more details of data sets and pattern sets.

For the large-gap pattern set (C_L), where $\Delta - \delta = 1000$ the optimal choice of b for all data sets is much higher—$b = 1024$ in all cases ($b = 512$ has very similar performance). Here we are seeing the effect of the time needed to set bits in the filter. For example, for Kernel, F already fits in L3 cache when $b = 8$, but at that setting $\lceil \frac{\Delta-\delta}{b} \rceil = 1000/8 = 125$ bits must be set in F per occurrence of p_1. With $b = 1024$ or 512, the number of bits set in F per occurrence of p_1 is just 1 or 2, the same as it is at the optimal setting for the small-gap (C_S) and middle-gap (C_M) pattern sets. This effect can probably be largely alleviated by employing two levels of filters or, alternatively, by implementing a method for setting a word of 1s at a time (effectively reducing the time to set bits from $\lceil \frac{\Delta-\delta}{b} \rceil$ to $\lceil \frac{\Delta-\delta}{b \cdot w} \rceil$, where w is the word size).

4 Direct Text Checking

The filtering ideas described in the previous section can drastically reduce the amount of time spent per subpattern occurrence, but the overall runtime is still $\Omega(occ_1 + occ_2)$, because both subpattern intervals are scanned in full. When the number of occurrences of the less frequent subpattern, say p_1, are significantly less than those of p_2, it is possible to get below that bound by scanning over only the occurrences of p_1, and for each occurrence, i, checking directly in the substring of text $\mathsf{T}[i + \delta..i + \Delta]$ for any occurrences of p_2, each of which corresponds to a match (or valid candidate match in the case $k > 2$). If we use a linear time pattern matching algorithm such as that by Knuth, Morris, and Pratt [13] to search for the occurrences of p_2, runtime (for two subpatterns) becomes $\Theta(occ_1 \cdot (\Delta - \delta))$.

Employed by itself, this kind of text checking can lead to terrible performance when both occ_1 and occ_2 are large. However, when employed in concert with a filter, it can lead to significant performance gains, particularly in later rounds of intersection when $k > 2$. Figure 4 illustrates this for $k = 2$, along with the performance of the other versions of SA-SCAN (Filter and Radix) we have decribed in previous sections. In sum, SA-SCAN has been sped up by more than an order of magnitude on some data sets. In Fig. 5 we see that the text checking heuristic makes an even bigger improvement when the number of subpatterns increases (from $k = 2$ to $k = 4$) because it is employed more often.

5 Experimental Evaluation

In this section we compare the practical performance of our version of SA-SCAN to the wavelet-tree-based method of Bader et al., which is called WT. We use a variety of texts and patterns, which are detailed below (most of these have appeared in experiments described in previous sections). Our methodology in this section closely follows that of [1].

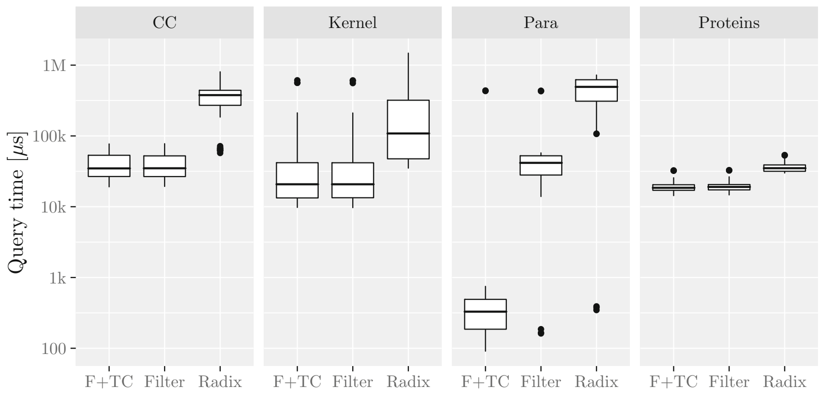

Fig. 4. Direct text checking improves search times further ($k = 2$).

Test Machine and Environment. We used a 2.10 GHz Intel Xeon E7-4830 v3 CPU equipped with 30 MiB L3 cache and 1.5 TiB of main memory. The machine had no other significant CPU tasks running and only a single thread of execution was used. The OS was Linux (Ubuntu 16.04, 64bit) running kernel 4.10.0-38-generic. Programs were compiled using `g++` version 5.4.0.

Texts. We use five datasets from different application domains:

- CC is a 2 GiB prefix of a recent 145TiB web crawl from commoncrawl.org.
- Kernel is a 2 GiB file consisting of source code of all (332) Linux kernel versions 2.2.X, 2.4.X.Y and 2.6.X.Y downloaded from kernel.org. The data set is very repetitive as only minor changes exist between subsequent versions.
- Para is a 410 MiB, which contains 36 sequences of Saccharomyces Paradoxus, is provided by the Saccharomyces Genome Resequencing Project. There are four bases $\{A, C, G, T\}$, but some characters denote an unknown choice among the four bases in which case N is used.
- Proteins is a 1.2 GiB sequence of newline-separated protein sequences (without descriptions, just the bare proteins) obtained from the Swissprot database. Each of the 20 amino acids is coded as one letter.

Patterns. As in [4], patterns were generated synthetically for each data set. We fixed the gap constraints $C_i = \delta_i, \Delta_i$ between subpatterns to small ($C_S = \langle 100, 110 \rangle$), medium ($C_M = \langle 1000, 1100 \rangle$), or large ($C_L = \langle 10000, 11000 \rangle$). VLG patterns were generated by extracting the 200 most common substrings of lengths 3, 5, and 7, which are then used as subpatterns. We then form 20 VLG patterns for each dataset, k (i.e. number of subpatterns), and gap constraint by selecting from the set of 200 subpatterns. We emphasise that the generated patterns, while not specifically designed to be pathological, do represent relatively hard instances for SA-scan because of the high frequency of each subpattern.

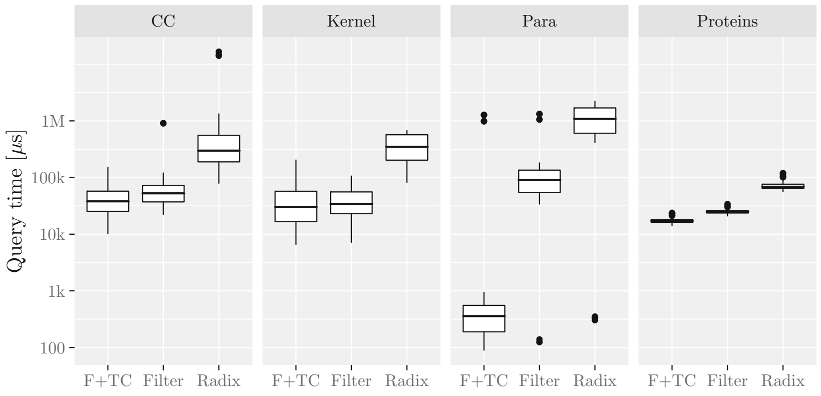

Fig. 5. Direct text checking improves search times further ($k = 4$).

Matching Performance for Different Gap Constraint Bands. Our first experiment aims to elucidate the impact of gap constraint size on query time. We fix the subpattern length $|p_i| = m_i = 3$. Table 1 shows the results from VLG patterns consisting of $k = 2^1, \ldots, 2^5$ subpatterns. Our method, marked FILTER+TC, is always faster than WT, with the exception of the large-gap C_L pattern sets, where on some data sets it yields to WT (most likely due to the text-checking heuristic being less effective on C_L).

Table 1. Total query time in milliseconds on all data sets for fixed $m_i = 3$ and gap constraints $C_S = \langle 100, 110 \rangle$, $C_M = \langle 1000, 1100 \rangle$, and $C_L = \langle 10000, 11000 \rangle$.

Method	CC			Kernel			Para			Proteins		
	C_S	C_M	C_L	C_S	C_M	C_L	C_S	C_M	C_L	C_S	C_M	C_L
k = 2												
Filter+TC	1110	1261	2106	739	823	1023	2372	4696	2335	393	523	1000
WT	14748	18066	41101	7763	8685	26982	14760	57026	99730	8812	11435	24922
k = 4												
Filter+TC	1420	1627	5941	420	1105	4341	5022	12742	16012	418	598	1589
WT	6458	6758	10582	1290	3821	5026	6578	48254	160223	8525	9463	15816
k = 8												
Filter+TC	1109	2857	5705	978	1107	1640	8708	16237	18845	400	597	2070
WT	4641	4358	5439	1255	520	1937	234	357	86996	12708	12866	14054
k = 16												
Filter+TC	1344	1989	4666	1581	1080	1646	3497	4802	13503	547	607	2313
WT	4410	5083	6224	527	513	326	262	260	253	20970	21731	23894
k = 32												
Filter+TC	1344	2176	5835	706	762	1749	6218	6171	18233	393	604	2335
WT	4532	6727	5722	491	668	568	500	540	527	45984	47297	50376

Matching Performance for Different Subpattern Lengths. In our second experiment, we examine the impact of subpattern lengths on query time, fixing the gap constraint to $C_S = 100, 110$. Table 2 shows the results. Larger subpattern lengths tend to result in smaller SA ranges. Consequently, SA-scan outperforms WT by an even wider margin.

Table 2. Total query time in milliseconds for fixed gap constraint $C_S = \langle 100, 110\rangle$ for different subpattern lengths $m_i \in \{3, 5, 7\}$ and different data sets.

Method	CC			Kernel			Para			Proteins		
	3	5	7	3	5	7	3	5	7	3	5	7
k = 2												
Filter+TC	1110	756	654	740	178	46	2372	641	78	393	32	22
WT	14748	8576	6158	7763	1731	93	14760	10176	2502	8812	441	182
k = 4												
Filter+TC	1420	362	310	420	159	53	5022	778	71	417	30	26
WT	6458	1477	637	1290	2182	30	6578	10882	2457	8525	97	67
k = 8												
Filter+TC	1109	683	230	978	206	196	8708	767	153	400	30	23
WT	4641	1380	464	1255	156	47	234	16558	3602	12708	51	33
k = 16												
Filter+TC	1344	541	679	1581	276	164	3497	836	77	547	31	32
WT	4410	922	412	527	155	81	262	29226	6317	20970	83	62
k = 32												
Filter+TC	1344	457	257	706	225	90	6218	730	234	393	33	61
WT	4532	1492	540	491	324	171	500	64070	13813	45984	177	128

Overall Runtime Performance. In a final experiment we explored the whole parameter space (i.e. $k \in \{2^1, \ldots, 2^5\}$, $m_i \in \{3, 5, 7\}$, $C \in \{C_S, C_M, C_L\}$). The results are summarized in Fig. 6. Overall out SA-scan-based method is faster on average than the wavelet-tree-based one, usually by a wide margin.

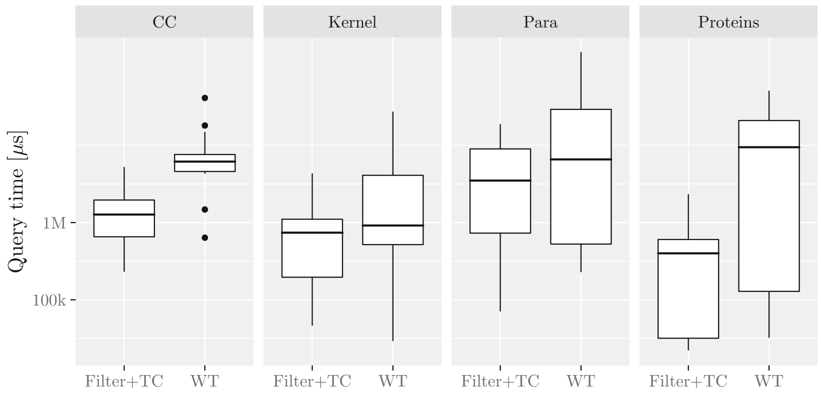

Fig. 6. Overall runtime performance of both methods, accumulating the performance for all $m_i \in \{3, 5, 7\}$ and C_S, C_M, and C_L.

6 Concluding Remarks

We have described a number of simple but highly effective improvements to the SA-scan VLG matching algorithm that, according to our experiments, elevate it to be the state-of-the-art approach for the indexed version of problem. We believe better indexing methods for VLG matching can be found, but that our version of SA-scan, which makes judicious use of filters, text checking, and subpattern processing order, represents a strong baseline against which the performance of more exotic methods should be measured.

Numerous avenues for continued work on VLG matching exist, perhaps the most interesting of which is to reduce index size. Currently, SA-scan uses $n \log n + n \log \sigma$ bits of space for a text of length n on alphabet σ for the suffix array and text, respectively (the WT approach of Bader et al., uses slightly more). Because our methods consist (mostly) of simple scans of SA ranges or scans of the underlying text, they are easily translated to make use of recent results on Burrows-Wheeler-based compressed indexes [10] that allow fast access to elements of the suffix array from a compressed representation of it. Via this observation we derive the first compressed indexes for VLG matching. These indexes use $O(r \log n)$ bits of space, where r is the number of runs in the Burrows-Wheeler transform, a quantity that decreases with text compressibility. On our 2 GiB Kernel data set, for example, the compressed index takes around 20 MiB in practice, and can still support VLG matching in times competitive with the indexes of Bader et al. We plan to explore this in more depth in future work.

Acknowledgments. Our thanks go to Tania Starikovskaya for suggesting the problem of indexing for regular-expression matching to us. We also thank Matthias Petri and Simon Gog for prompt answers to questions about their article and code and the anonymous reviewers for helpful comments. This work was funded by the Academy of Finland via grant 319454 and by EU's Horizon 2020 research and innovation programme under Marie Skłodowska-Curie grant agreement No. 690941 (BIRDS).

References

1. Bader, J., Gog, S., Petri, M.: Practical variable length gap pattern matching. In: Goldberg, A.V., Kulikov, A.S. (eds.) SEA 2016. LNCS, vol. 9685, pp. 1–16. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-38851-9_1
2. Bille, P., Farach-Colton, M.: Fast and compact regular expression matching. Theor. Comput. Sci. **409**(3), 486–496 (2008)
3. Bille, P., Gørtz, I.L.: Substring range reporting. Algorithmica **69**(2), 384–396 (2014)
4. Bille, P., Gørtz, I.L., Vildhøj, H.W., Wind, D.K.: String matching with variable length gaps. Theor. Comput. Sci. **443**, 25–34 (2012)
5. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: Proceedings of SODA, pp. 1297–1308. ACM-SIAM (2010)
6. Cox, R.: Regular expression matching with a trigram index or how Google code search worked (2012). https://swtch.com/~rsc/regexp/regexp4.html

7. Crawford, T., Iliopoulos, C.S., Raman, R.: String matching techniques for musical similarity and melodic recognition. Comput. Musicol. **11**, 73–100 (1998)
8. Crochemore, M., Iliopoulos, C.S., Makris, C., Rytter, W., Tsakalidis, A.K., Tsichlas, T.: Approximate string matching with gaps. N. J. Comput. **9**(1), 54–65 (2002)
9. Fredriksson, K., Grabowski, S.: Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. Inf. Retr. **11**(4), 335–357 (2008)
10. Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in BWT-runs bounded space. In: Proceedings of SODA, pp. 1459–1477. ACM-SIAM (2018)
11. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proceedings of the SODA, pp. 841–850. ACM-SIAM (2003)
12. Haapasalo, T., Silvasti, P., Sippu, S., Soisalon-Soininen, E.: Online dictionary matching with variable-length gaps. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 76–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20662-7_7
13. Knuth, D., Morris, J.H., Pratt, V.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
14. Lewenstein, M.: Indexing with gaps. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 135–143. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24583-1_14
15. Lopez, A.: Hierarchical phrase-based translation with suffix arrays. In: Proceedings of the EMNLP-CoNLL 2007, pp. 976–985. ACL (2007)
16. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
17. Metzler, D., Croft, W.B.: A markov random field model for term dependencies. In: Proceedings of the SIGIR, pp. 472–479. ACM (2005)
18. Morgante, M., Policriti, A., Vitacolonna, N., Zuccolo, A.: Structured motifs search. J. Comput. Biol. **12**(8), 1065–1082 (2005)
19. Navarro, G.: Wavelet trees for all. J. Discrete Algorithms **25**, 2–20 (2014)
20. Pissis, S.P.: MoTeX-II: structured MoTif eXtraction from large-scale datasets. BMC Bioinform. **15**(235), 1–12 (2014)
21. Saikkonen, R., Sippu, S., Soisalon-Soininen, E.: Experimental analysis of an online dictionary matching algorithm for regular expressions with gaps. In: Bampis, E. (ed.) SEA 2015. LNCS, vol. 9125, pp. 327–338. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20086-6_25
22. Turpin, A., Tsegay, Y., Hawking, D., Williams, H.E.: Fast generation of result snippets in web search. In: Proceedings of the SIGIR 2007, pp. 127–134. ACM (2007)