Practical Variable Length Gap Pattern Matching

Johannes Bader[1], Simon Gog[1(✉)], and Matthias Petri[2]

[1] Institute of Theoretical Informatics, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
gog@kit.edu

[2] Department of Computing and Information Systems, The University of Melbourne, VIC 3010, Australia

Abstract. Solving the problem of reporting all occurrences of patterns containing variable length gaps in an input text T efficiently is important for various applications in a broad range of domains such as Bioinformatics or Natural Language Processing. In this paper we present an efficient solution for static inputs which utilizes the wavelet tree of the suffix array. The algorithm partially traverses the wavelet tree to find matches and can be easily adapted to several variants of the problem. We explore the practical properties of our solution in an experimental study where we compare to online and semi-indexed solutions using standard datasets. The experiments show that our approach is the best choice for searching patterns with many gaps in large texts.

1 Introduction

The classical *pattern matching problem* is to find all occurrences of a pattern $\mathcal{P}$ (of length m) in a text $\mathcal{T}$ (of length n both drawn from an alphabet Σ of size σ). The *online* algorithm of Knuth, Morris and Pratt [10] utilizes a precomputed table over the pattern to solve the problem in $\mathcal{O}(n+m)$ time. Precomputed indexes over the input text such as *suffix arrays* [14] or *suffix trees* allow matching in $\mathcal{O}(m \times \log n)$ or $\mathcal{O}(m)$ time respectively. In this paper we consider the more general *variable length gap pattern matching problem* in which a pattern does not only consist of characters but also length constrained *gaps* which match any character in the text. We formally define the problem as:

Problem 1 Variable Length Gap (VLG) Pattern Matching [5]. Let $\mathcal{P}$ be a pattern consisting of $k \geq 2$ subpatterns $p_0 \dots p_{k-1}$, of lengths $m = m_0 \dots m_{k-1}$ drawn from Σ and $k-1$ gap constraints $C_0 \dots C_{k-2}$ 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) distance between a match of p_i and p_{i+1} in $\mathcal{T}$. Find all matches – given as k-tuples $\langle i_0 \dots i_{k-1} \rangle$ where i_j is the starting position for subpattern p_j in $\mathcal{T}$ – such that all gap constraints are satisfied.

If overlaps between matches are permitted, the number of matching positions can be polynomial in $k+1$. We refer to this problem type as *all*. In standard

© Springer International Publishing Switzerland 2016
A.V. Goldberg and A.S. Kulikov (Eds.): SEA 2016, LNCS 9685, pp. 1–16, 2016.
DOI: 10.1007/978-3-319-38851-9_1

implementations of *regular expressions* overlaps are not permitted[1] and two types for variable gap pattern matching are supported. The *greedy* type (a gap constraint is written as $C_i = \texttt{.}\{\delta_i, \Delta_i\}$) maximizes while the *lazy* type (a gap constraint is written as $C_i = \texttt{.}\{\delta_i, \Delta_i\}\texttt{?}$) minimizes the characters matched by gaps. The following example illustrates the three VLG problem types.

Example 1. *Given a text $\mathcal{T}$ = $\texttt{aaabbbbaaabbbb}$ and a pattern $\mathcal{P} = \texttt{ab}\langle 1,6\rangle\texttt{b}$ consisting of two subpatterns p_0 = "ab" and p_1 = "b" and a gap constraint $C_0 = \langle 1,6\rangle$. Type "all" returns $S_{all} = \{\langle 2,5\rangle, \langle 2,6\rangle, \langle 2,10\rangle, \langle 9,12\rangle, \langle 9,13\rangle\}$, greedy matching results in $S_{greedy} = \{\langle 2,10\rangle\}$ and lazy evaluation $S_{lazy} = \{\langle 2,5\rangle, \langle 9,12\rangle\}$.*

VLG pattern matching is an important problem which has numerous practical applications. Traditional Unix utilities such as `grep` or `mutt` support VLG pattern matching on small inputs using regular expression engines. Many areas of computer science use VLG matching on potentially very large data sets. In Bioinformatics, Navarro and Raffinot [18] investigate performing VLG pattern matching in protein databases such as PROSITE [9] where individual protein site descriptions are expressed as patterns containing variable length gaps. In Information Retrieval (IR), the proximity of query tokens within a document can be an indicator of relevance. Metzler and Croft [15] define a language model which requires finding query terms to occur within a certain window of each other in documents. In Natural Language Processing (NLP), this concept is often referred to as collocations of words. Collocations model syntactic elements or semantic links between words in tasks such as word sense disambiguation [16]. In Machine Translation (MT) systems VLG pattern matching is employed to find translation rule sets in large text corpora to improve the quality of automated language translation systems [13].

In this paper, we focus on the *offline* version of the VLG pattern matching problem. Here, a static input is preprocessed to generate an *index* which facilitates faster query processing. Our contributions are as follows:

1. We build an index consisting of the wavelet tree over the suffix array and propose different algorithms to efficiently answer VLG matching queries. The core algorithms is conceptionally simple and can be easily adjusted to the three different matching modes outlined above.
2. In essence our WT algorithm is faster than other intersection based approaches as it allows to combine the sorting and filtering step and does not require copying of data. Therefore our approach is specially suited for a large number of subpatterns.
3. We provide a thorough empirical evaluation of our method including a comparison to different practical baselines including other index based approaches like qgram indexes and suffix arrays.

[1] I.e. any two match tuples $\langle i_0 \dots i_{k-1}\rangle$ and $\langle i'_0 \dots i'_{k-1}\rangle$ spanning the intervals $[i_0, i_{k-1} + m_{k-1} - 1]$ and $[i'_0, i'_{k-1} + m_{k-1} - 1]$ do not overlap.

2 Background and Related Work

Existing solutions to solving VLG can be categorized into three classes of algorithms. In general the algorithms discussed here perform lazy evaluation, but can be implemented to also support greedy evaluation. The first category of algorithms build on the classical algorithm of Thompson [20] to construct a finite automaton to solve the VLG problem used by many regular expression engines. Let $L = \sum_{i=0}^{k-2} \Delta_i$. The classical automaton requires $\mathcal{O}\big(n(L\sigma + m)\big)$ time which can not be reduced much further [4,5]. The matching process scans $\mathcal{T}$, transitioning between states in the automaton to find occurrences of $\mathcal{P}$. Algorithms engineered to solve the VLG problem specifically can achieve better runtime performance by utilizing bit-parallelism or placing constraints on individual gap constraints in $\mathcal{P}$ [4,6,18]. For example the runtime of Bille and Thorup [4] is $\mathcal{O}\big(n(k\frac{\log w}{w} \times \log k)\big)$ time after prepocessing $\mathcal{P}$ (w is the word size).

A second class of algorithms take into account the occurrences of each subpattern in $\mathcal{P}$ [5,17,19]. The algorithms operate in two stages. First, all occurrences of each subpattern $p \in \mathcal{P}$ in $\mathcal{T}$ are determined. Let α_i be the number of occurrences of p_i in $\mathcal{T}$ and $\alpha = \sum_{i=0}^{k-1} \alpha_i$ be the total number of occurrences of all p_i in $\mathcal{T}$. The occurrences of each subpattern can be obtained via a classical online algorithm such as Aho and Corasick [1] (AC), or using an index such as the suffix array (SA) of $\mathcal{T}$. The algorithms of Morgante et al. [17,19] require additional $\mathcal{O}\big(\alpha\big)$ space to store the occurrences, whereas Bille et al. [5] only requires $\mathcal{O}\big(S\big)$ extra space where $S = \sum_{i=0}^{k-2} \delta_i$. The algorithms keep track of, for each p_{i+1} the occurrences of p_i for which C_i is satisfied. Similarly to Rahman et al. [19], the occurrences $X_i = [x_0, \ldots, x_{\alpha_i - 1}]$ of p_i are used to satisfy $C_i = \langle \delta_i, \Delta_i \rangle$ by searching for the next larger (or equal) value for all $x_j + \delta_i$ in the sorted list of occurrences of p_{i+1}. Performing this process for all p_i and gap constraints C_i can be used to perform lazy evaluation of VGP. While Rahman et al. [19] consider only AC or SA to identify occurrences of subpatterns, many different ways to store or determine and search positions exist. For example, the folklore q-gram index, which explicitly stores the occurrences of all q-grams in $\mathcal{T}$, can be used to obtain the occurrences of all subpatterns by performing intersection of the positional lists of the q-grams each subpattern in $\mathcal{P}$. List compression affects the performance of the required list intersections and thus provides different time space-trade-offs [11]. Similarly, different list intersection algorithms can also affect the performance of such a scheme [2].

A combination of schemata one and two use occurrences of subpatterns to restrict the segments within $\mathcal{T}$ where efficient *online* algorithms are used to *verify* potential matches. For example, q-gram lists can be intersected until the number of possible occurrences of $\mathcal{P}$ is below a certain threshold. Then an automaton based algorithm can match $\mathcal{P}$ at these locations.

The third category are suffix tree based indexes. In its simplest form, Lewenstein [12] augments each node of a suffix tree over $\mathcal{T}$ with multiple gap-r-tree for all $1 \leq r < G$, where G is the longest gap length which has to be specified at construction time. If k subpatterns are to be supported, the nodes in each gap-r-tree have to be recursively augmented with additional gap-r-trees at a total

space cost of $\mathcal{O}(n^k G^{k-1})$ space. Queries are answered in $\mathcal{O}(\sum_0^{k-1} m_i)$ time by traversing the suffix tree from the root, branching into the gapped-r-trees after the locus of p_0 is found. Lewenstein [12] further propose different time-space trade-offs, reducing the space to $\mathcal{O}(nG^{2k-1}\log^{k-1} n)$ by performing centroid path decomposition which increases query time to $\mathcal{O}(\sum_0^{k-1} m_i + 2^{k-1}\log\log n)$. Bille and Gørtz [3] propose a suffix tree based index which requires two ST over $\mathcal{T}$ ($ST(\mathcal{T})$) and the reverse text ($ST(\mathcal{T}^R)$) plus a range reporting structure. Single fixed length gap queries can then be answered by matching p_0 in $ST(\mathcal{T})$ and the reverse of p_1 in $ST(\mathcal{T}^R)$. Then a range reporting structure is used to find the matching positions in $\mathcal{T}$.

In practice, Lopez [13] use a combination of (1) intersection precomputation (2) fast list intersection and (3) enhanced version of Rahman et al. [19] to solve a restricted version of VGP.

3 VLG Pattern Matching Using the Wavelet Tree over SA

We first introduce notation which is necessary to describe our algorithms. Let range I from index ℓ to r be denoted by $[\ell, r]$. A range is considered empty ($I = \emptyset$) if $\ell > r$. We denote the intersection of two ranges $I_0 = [\ell_0, r_0]$ and $I_1 = [\ell_1, r_1]$ as $I_0 \cap I_1 = [\max\{\ell_0, \ell_1\}, \min\{r_0, r_1\}]$. We further define the addition $I_0 + I_1$ of two ranges to be $[\ell_0 + \ell_1, r_0 + r_1]$. Shorthands for the left and right border of a non-empty range I are $lb(I)$ and $rb(I)$.

Let $X_i = x_{i,0}, \ldots, x_{i,\alpha_i-1}$ be the list of starting positions of subpattern p_i in $\mathcal{T}$. Then for $k = 2$ the solution to the VLG pattern matching problem for $\mathcal{P} = p_0\langle\delta, \Delta\rangle p_1$ are pairs $\langle x_{0,i}, x_{1,j}\rangle$ such that $([x_{0,i}, x_{0,i}] + [m_0 + \delta, m_0 + \Delta]) \cap [x_{1,j}, x_{1,j}] \neq \emptyset$. The generalization to $k > 2$ subpatterns is straightforward by checking all $k-1$ constraints. For ease of presentation we will restrict the following explanation to $k = 2$. Assuming all X_i are present in sorted order all matches can be found in $\mathcal{O}(\alpha_0 + \alpha_1 + z)$ time, where z refers to the number of matches of $\mathcal{P}$ in $\mathcal{T}$. Unfortunately, memory restrictions prohibit the storage of all possible $\mathcal{O}(n^2)$ sorted subpattern lists, but we will see next that the linear space suffix array can be used to retrieve any unsorted subpattern list.

A suffix $\mathcal{T}[i, n-1]$ is identified by its starting position i in $\mathcal{T}$. The suffix array (SA) contains all suffixes in lexicographically sorted order, i.e. SA[0] points to the smallest suffix in the text, SA[1] to the second smallest and so on. Figure 1 depicts the SA for an example text of size $n = 32$. Using SA and $\mathcal{T}$ it is easy to determine all suffixes which start with a certain prefix p by performing binary search. For example, $p_0 =$ `gt` corresponds to the SA-interval $[17, 21]$ which contains suffixes $16, 13, 21, 4, 28$. Note that the occurrences of the p in SA are not stored sorted order. Answering a VLG pattern query using SA can be achieved by first determining the SA-intervals of all subpatterns and next, filtering out all occurrence tuples which fulfill the gap constraints [19].

Let $\mathcal{P} =$ `gc`$\langle 1, 2\rangle$`c` containing $p_0 =$ `gc` and $p_1 =$ `c`. In Example Fig. 1, the SA-interval of `c` (SA$[9, 16]$) contains suffixes $26, 10, 11, 19, 12, 20, 1$ and 8.

$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\mathcal{T} =$	a	c	t	a	g	t	a	t	c	t	c	c	c	g	t	a	g	t	a	c	c	g	t	a	t	a	c	a	g	t	t	$
SA($\mathcal{T}$) =	31	25	18	0	15	3	27	23	6	26	10	11	19	12	20	1	8	16	13	21	4	28	30	24	17	14	2	22	5	9	7	29

Fig. 1. Sample text $\mathcal{T}$ = `actagtatctcccgtagtaccgtatacagtt$` and suffix array (SA) of $\mathcal{T}$.

Sorting the occurrences of both subpatterns returns in $X_0 = 4, 13, 16, 21, 28$ and $X_1 = 1, 8, 10, 11, 12, 19, 20, 26$. Filtering X_0 and X_1 based on $C_0 = \langle 1, 2\rangle$ produces tuples $\langle 4, 8\rangle, \langle 16, 19\rangle$ and $\langle 16, 20\rangle$. The time complexity of this process is $\mathcal{O}\big(\sum_{i=0}^{k-1} \alpha_i \log \alpha_i + z\big)$, where the first term (sorting all X_i) is independent of z (the output size) and can dominate if subpatterns occur frequently.

Using a wavelet tree (WT) [8] allows *combining the sorting and filtering process*. This enables *early termination* for text regions which do not contain all required subpatterns in correct order within the specified gap constraints. A wavelet tree $WT(X)$ of a sequence $X[0, n-1]$ over an alphabet $\Sigma[0, \sigma-1]$ is defined as a perfectly balanced binary tree of height $H = \lceil \log \sigma \rceil$. Conceptually the root node v represents the whole sequence $X_v = X$. The left (right) child of the root represents the subsequence $X_{\mathtt{0}}$ ($X_{\mathtt{1}}$) which is formed by only considering symbols of X which are prefixed by a `0`-bit(`1`-bit). In general the i-th node on level L represents the subsequence $X_{i_{(2)}}$ of X which consists of all symbols which are prefixed by the length L binary string $i_{(2)}$. More precisely the symbols in the range $R(v_{i_{(2)}}) = [i \cdot 2^{H-L}, (i+1) \cdot 2^{H-L} - 1]$. Figure 2 depicts an example for $X = \mathrm{SA}(\mathcal{T})$. Instead of actually storing $X_{i_{(2)}}$ it is sufficient to store the bitvector $B_{i_{(2)}}$ which consists of the ℓ-th bits of $X_{i_{(2)}}$. In connection with a *rank structure*, which can answer how many `1`-bits occur in a prefix $B[0, j-1]$ of bitvector $B[0, n-1]$ in constant time using only $o(n)$ extra bits, one is able to reconstruct all elements in an arbitrary interval $[\ell, r]$: The number of `0`-bits (`1`-bits) left to ℓ corresponds to ℓ' in the left (right) child and the number of `0`-bits (`1`-bits) left to r corresponds to $r' + 1$ in the left (right) child. Figure 2 shows this *expand* method. The red interval $[17, 21]$ in the root node v is expanded to $[9, 10]$ in node v_0 and $[8, 10]$ in node v_1. Then to $[4, 4]$ in node v_{00} and $[5, 5]$ in node v_{01} and so on. Note that WT nodes are only traversed if the interval is not empty (i.e. $\ell \le r$). E.g. $[4, 4]$ at v_{00} is split into $[3, 2]$ and $[1, 1]$. So the left child v_{000} is omitted and the traversal continues with node v_{001}. Once a leaf is reached we can output the element corresponding to its root to leaf path. The wavelet tree $WT(X)$ uses just $n \cdot h + o(n \cdot h)$ bits of space.

In our application the initial intervals correspond to the SA-intervals of all subpatterns p_i in $\mathcal{P}$. However, our traversal algorithm only considers the existence of a SA-interval at a given node and not its size. A non-empty SA-interval of subpattern p_i in a node v_x at level L means that p_i occurs somewhere in the text range $R(v_x) = [x \cdot 2^{H-L}, (x+1) \cdot 2^{H-L} - 1]$. Figure 3 shows the text ranges for each WT node. A node v and its parent edge is marked red (resp. blue) if subpattern p_0's (resp. p_1's) occurs in the text range $R(v)$.

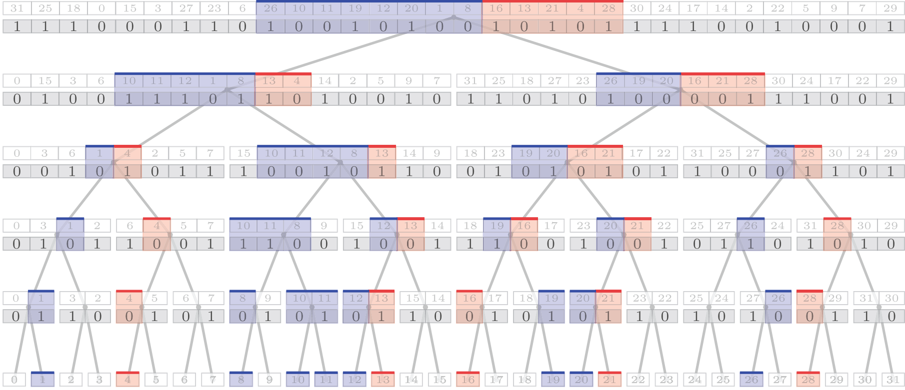

Fig. 2. Wavelet tree built for the suffix array of our example text. The SA-interval of `gt` (resp. `c`) in the root and its expanded intervals in the remaining WT nodes are marked red (resp. blue). (Color figure online)

3.1 Breadth-First Search Approach

For both subpatterns p_0 and p_1, at each level in the WT, j we iteratively materialize lists N_0^j and N_1^j of all WT nodes at level j in which the ranges corresponding to p_0 and p_1 occur by expanding the nodes in the lists N_0^{j-1} and N_1^{j-1} of the previous level. Next all nodes v_xs in N_0^j are removed if there is no node v_y in N_1^j such that $(R(v_x) + [m_0 + \delta, m_0 + \Delta]) \cap [R(v_y)] \neq \emptyset$ and vice versa. Each list N_i^{j-1} stores nodes in sorted order according to the beginning of their ranges. Thus, removing all "invalid" nodes can be performed in $\mathcal{O}\big(|N_0^j| + |N_1^j|\big)$ time. The following table shows the already filtered list for our running example.

WT level (j)	p_0 text ranges (N_0^j)	p_1 text ranges (N_1^j)
0	$[0, 31]$	$[0, 31]$
1	$[0, 15], [16, 31]$	$[0, 15], [16, 31]$
2	$[0, 7], [8, 15], [16, 23], [24, 31]$	$[0, 7], [8, 15], [16, 23], [24, 31]$
3	$[4, 7], [12, 15], [16, 19], [20, 23]$	$[8, 11], [12, 15], [16, 19], [20, 23], [24, 27]$
4	$[4, 5], [16, 17], [20, 21]$	$[8, 9], [18, 19], [20, 21], [24, 25]$
5	$[4, 4], [16, 16]$	$[8, 8], [19, 19], [20, 20]$

The WT nodes in the table are identified by their text range as shown in Fig. 3. One example of a removed node is $[0, 3]$ in lists N_0^3 and N_1^3 which was expanded from node $[0, 7]$ in N_0^2 and N_1^2. It was removed since there is no text range in N_1^3 which overlaps with $[0, 3] + [2 + 1, 2 + 2] = [3, 6]$. Figure 3 connects removed WT nodes with dashed instead of solid edges. Note that all text positions at the leaf level are the start of a subpattern which fulfills all gap constraints. For

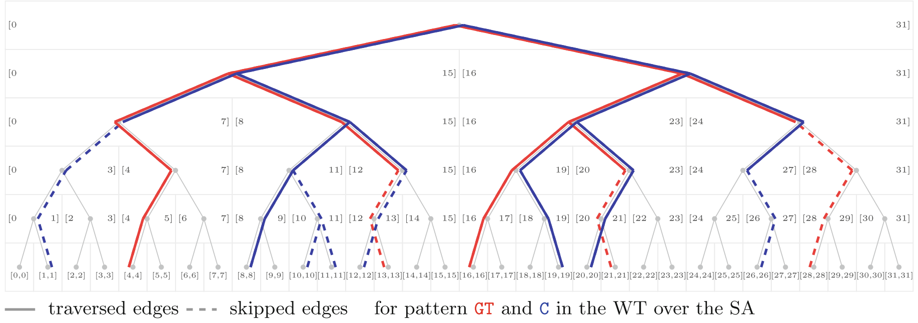

Fig. 3. Wavelet tree nodes with annotated text ranges and path of subpattern iterators. (Color figure online)

the *all* variant it just takes $\mathcal{O}(z)$ time to output the result. The disadvantage of this BFS approach is that the lists of a whole level have to be kept in memory, which takes up to n words of space. We will see next that a DFS approach lowers memory consumption to $\mathcal{O}(k \log n)$ words.

3.2 Depth-First Search Approach

For each subpattern p_i we create a depth-first search *iterator* it_i. The iterator consists of a stack of (WT node, SA-interval) pairs, which is initialized by the WT root node and the SA-interval of p_i, in case the SA-interval is not empty. The iterator is invalid, if the stack is empty – this can be checked in constant time using a method $valid(it_i)$. We refer with $it_i.v$ to the current WT node of a valid iterator (which is on top of the stack). A valid iterator can be incremented by operations *next_down* and *next_right*. Method *next_down* pops pair $(v, [\ell, r])$, expands SA-interval $[\ell, r]$ and pushes the right child of v with its SA-interval and the left child of v with its SA-interval onto the stack, if the SA-interval is not empty. That is, we traverse to the leftmost child of $it.v$ which contains p_i. The $next_right(it_i)$ operation pops one element from the stack, i.e. we traverse to the leftmost node in the WT which contains p_i and is right of $it_i.v$.

Using these iterators the VLG pattern matching process can be expressed succinctly in Algorithm 1, which reports the next match. The first line checks, if both iterators are still valid so that a further match can be reported. Lines 2 and 4 check if the gap constraints are met. If the text range of p_1's iterator is too far right (Line 2), the iterator of p_0 is moved right in Line 3. Analogously, the iterator of p_1 is moved right in Line 5 if the text range of p_0's iterator is too far right (Line 4). If the gap constrained is met and not both text ranges have size one (Line 7) we take the iterator which is closer to the root (and break ties by i) and refine its range. Finally, if both iterators reach the leaf level a match can be reported. Since the traversal finds the two leftmost leaf nodes – i.e. positions – which met the constraint the direct output of $\langle lb(R(it_0.v)), lb(R(it_1.v))\rangle$ in Line 11 corresponds to the *lazy* problem type. For *lazy* Line 12 would move it_0 to the

Algorithm 1. dfs_next_match($it_0, it_1, m_0, \Delta_0, \delta_0$)

```
1: while valid(it_0) and valid(it_1) do
2:     if rb(R(it_0.v)) +m_0+ Δ_0 < lb(R(it_1.v)) then      # gap constraint violated?
3:         it_0 ← next_right(it_0)
4:     else if rb(R(it_1.v)) < lb(R(it_0.v)) + m_0 + δ_0 then  # gap constraint violated?
5:         it_1 ← next_right(it_1)
6:     else                                                # gap constrained fulfilled
7:         if not (is_leaf(it_0.v) and is_leaf(it_1.v)) then
8:             x ← arg min_{i∈{0,1}} {⟨depth(it_i.v), i⟩}     # select itr closest to the root
9:             it_x ← next_down(it_x)                         # refine range
10:        else
11:            report match according to VLG problem type    # found match
12:            move it_0 and it_1 according to VLG problem type and return ⟨it_0, it_1⟩
```

right of it_1 by calling $it_0 \leftarrow next_right(it_0)$ until $lb(R(it_0.v)) > lb(R(it_1.v))$ is true and no overlapped matches are possible. Type *greedy* can be implemented by moving it_1 in Line 11 as far right as possible within the gap constrains, output $\langle lb(R(it_0.v)), lb(R(it_1.v))\rangle$, and again moving it_0 to the right of it_1. Type *all* reports the first match in Line 11, then iterates it_1 as long as it meets the gap constraint and reports a match if $it_1.v$ is a leaf. In Line 12 it_0 is move one step to the right and it_1 it reset to its state before line 11.

3.3 Implementation Details

Our representation of the WT requires two rank operations to retrieve the two child nodes of any tree node. In our DFS approach, k tree iterators partially traverse the WT. For higher values of k it is likely that the child nodes of a specific WT node are retrieved multiple times by different iterators. We therefore examined the effect of caching the child nodes of a tree node when they are retrieved for the first time, so any subsequent child retrieval operations can be answered without performing further rank operations. Unfortunately, this approach resulted in a slowdown of our algorithm by a factor of 3. We conjecture, that one reason for this slowdown is the additional memory management overhead (even when using custom allocators) of dynamically allocating and releasing the cached data. Also, critical portions of the algorithm (being called most frequently) contain more branching and were even inlined before we implemented the cache. Furthermore, we determined that more than 65 % of tree nodes traversed once were never traversed a second time, so caching children for these nodes will not yield any run time performance improvements. On average, each cache entry was accessed less than 2 times after creation. Thus, only very few rank operations are actually saved. Therefore we do not cache child nodes in our subsequent empirical evaluation.

4 Empirical Evaluation

In this section we study the practical impact of our proposals by comparing to standard baselines in different scenarios. Our source code – including baselines and dataset details – is publicly available at https://github.com/olydis/vlg_matching and implemented on top of SDSL [7] data structures. We use three datasets from different application domains:

- The CC data set is a 371 GiB prefix of a recent 145 TiB web crawl from commoncrawl.org.
- The Kernel data set is a 78 GiB file consisting of the source code of all (332) Linux kernel versions 2.2.X, 2.4.X.Y and 2.6.X.Y downloaded from kernel.org. This data set is very repetitive as only minor changes exist between subsequent kernel versions.
- The Dna-Hg38 data set data consisting of the 3.1 GiB Genome Reference Consortium Human Reference 38 in fasta format with all symbol $\notin \{A, C, G, T\}$ removed from the sequence.

We have implemented our BFS and DFS wavelet tree approaches. We omit the results of the BFS approach, as DFS dominated BFS in both query time and memory requirement. Our index is denoted by WT-DFS the following. We use a pointerless WT (`wt_int`) in combination with a fast rank enabled bitvector (`bit_vector_il`). We compare to three baseline implementations:

- RGXP: A "off-the-shelf" automaton based regular expression engine (BOOST library version 1.58; `ECMAScript` flag set) which scans the whole text.
- QGRAM-RGXP: A q-gram index ($q = 3$) which stores absolute positions of all unique 3-grams in the text using Elias-Fano encoding. List intersection is used to produce candidate positions in $\mathcal{T}$ and subsequently checked by the RGXP engine.
- SA-SCAN: The plain SA is used as index. The SA-intervals of the subpatterns are determined, sorted, and filtered as described in earlier. This approach is similar to that of Rahman et al. [19] while replacing the van Emde Boas tree by sorting ranges.

All baselines and indexes are implemented using `C++11` and compiled using `gcc 4.9.1` using all optimizations. The experiments were performed on a machine with an Intel Xeon E4640 CPU and 148 GiB RAM. The default VLG matching type in our experiments is *lazy*, which is best suited for proximity search. Pattern were generated systematically for each data set. We fix the gap constraints $C_i = \langle \delta_i, \Delta_i \rangle$ between subpatterns to $\langle 100, 110 \rangle$ small (C_S), $\langle 1\,000, 1\,100 \rangle$ medium (C_M), or $\langle 10\,000, 11\,000 \rangle$ large (C_L). For each dataset we extract the 200 most common subpatterns of length 3, 5 and 7 (if possible). We form 20 regular expressions for each dataset, k, and gap constraint by selecting from the set of subpatterns.

Matching Performance for Different Gap Constraint Bands. In our first experiment we measure the impact of gap constraint size on query time. We

Table 1. Median query time in milliseconds for fixed $m_i = 3$ and text size 2 GiB for different gap constraints $\langle 100, 110\rangle$ small (C_S), $\langle 1000, 1100\rangle$ medium (C_M) or $\langle 10\,000, 11\,000\rangle$ large (C_L) and three data sets.

Method	Kernel-2G			CC-2G			Dna-Hg38-2G		
	C_S	C_M	C_L	C_S	C_M	C_L	C_S	C_M	C_L
$k = 2$									
RGXP	6 383	7 891	18 592	2 533	4 148	17 394	24 363	26 664	9 849
QGRAM-RGXP	695	2 908	20 775	650	2 604	21 027	48 984	33 911	7 711
SA-SCAN	115	114	113	132	130	132	6 762	6 661	6 433
WT-DFS	279	244	347	180	211	277	17 041	10 978	8 350
$k = 4$									
RGXP	5 130	6 840	30 948	5 076	6 889	28 931	34 025	41 800	24 549
QGRAM-RGXP	1 336	8 992	$\geq 10^5$	1 284	9 187	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$	91 137
SA-SCAN	247	249	250	284	284	289	14 667	14 971	14 191
WT-DFS	160	164	183	195	201	232	19 977	12 608	8 506
$k = 8$									
RGXP	3 243	5 089	31 796	2 426	4 215	28 943	33 126	$\geq 10^5$	$\geq 10^5$
QGRAM-RGXP	3 307	30 174	$\geq 10^5$	2 894	27 488	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$
SA-SCAN	594	585	596	759	761	765	29 850	30 621	29 296
WT-DFS	263	282	228	184	185	179	28 343	16 707	8 843
$k = 16$									
RGXP	3 447	5 278	32 782	2 407	4 229	33 828	37 564	$\geq 10^5$	$\geq 10^5$
QGRAM-RGXP	6 843	61 787	$\geq 10^5$	5 967	65 722	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$
SA-SCAN	1 400	1 402	1 416	1 714	1 711	1 690	56 558	62 423	55 017
WT-DFS	508	507	463	331	331	316	55 660	26 041	9 152
$k = 32$									
RGXP	3 446	5 237	32 979	3 673	6 041	33 957	24 040	$\geq 10^5$	$\geq 10^5$
QGRAM-RGXP	14 732	$\geq 10^5$	$\geq 10^5$	11 506	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$
SA-SCAN	2 885	2 926	2 924	3 573	3 560	3 562	82 663	92 756	81 164
WT-DFS	1 183	1 083	965	614	609	594	35 495	35 212	5 501

fix the dataset size to 2 GiB and the subpattern length $|p_i| = m_i = 3$; Table 1 shows the results for pattern consisting of $k = 2^1, \ldots, 2^5$ subpatterns. For RGXP, the complete text is scanned for all bands. However, the size of the underlying automaton increases with the gap length. Thus, the performance decreases for larger gaps. The intersection process in QGRAM-RGXP reduces the search space of RGXP to a portion of the text. There are cases where the search space reduction is not significant enough to amortize the overhead of the intersection. For example, the large gaps or the small alphabet test case force QGRAM-RGXP to perform more work than RGXP. The two SA based solutions, SA-SCAN and WT-DFS, are

Table 2. Median query time in milliseconds for fixed gap constraint $\langle 100, 110\rangle$ and text size 2 GiB for different subpattern lengths $m_i \in 3, 5, 7$ for three data sets.

Method	Kernel-2G			CC-2G			Dna-Hg38-2G		
	C_S	C_M	C_L	C_S	C_M	C_L	C_S	C_M	C_L
$k = 2$									
RGXP	6 383	7 891	18 592	2 533	4 148	17 394	24 363	26 664	9 849
QGRAM-RGXP	695	2 908	20 775	650	2 604	21 027	48 984	33 911	7 711
SA-SCAN	115	114	113	132	130	132	6 762	6 661	6 433
WT-DFS	279	244	347	180	211	277	17 041	10 978	8 350
$k = 4$									
RGXP	5 130	6 840	30 948	5 076	6 889	28 931	34 025	41 800	24 549
QGRAM-RGXP	1 336	8 992	$\geq 10^5$	1 284	9 187	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$	91 137
SA-SCAN	247	249	250	284	284	289	14 667	14 971	14 191
WT-DFS	160	164	183	195	201	232	19 977	12 608	8 506
$k = 8$									
RGXP	3 243	5 089	31 796	2 426	4 215	28 943	33 126	$\geq 10^5$	$\geq 10^5$
QGRAM-RGXP	3 307	30 174	$\geq 10^5$	2 894	27 488	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$
SA-SCAN	594	585	596	759	761	765	29 850	30 621	29 296
WT-DFS	263	282	228	184	185	179	28 343	16 707	8 843
$k = 16$									
RGXP	3 447	5 278	32 782	2 407	4 229	33 828	37 564	$\geq 10^5$	$\geq 10^5$
QGRAM-RGXP	6 843	61 787	$\geq 10^5$	5 967	65 722	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$
SA-SCAN	1 400	1 402	1 416	1 714	1 711	1 690	56 558	62 423	55 017
WT-DFS	508	507	463	331	331	316	55 660	26 041	9 152
$k = 32$									
RGXP	3 446	5 237	32 979	3 673	6 041	33 957	24 040	$\geq 10^5$	$\geq 10^5$
QGRAM-RGXP	14 732	$\geq 10^5$	$\geq 10^5$	11 506	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$	$\geq 10^5$
SA-SCAN	2 885	2 926	2 924	3 573	3 560	3 562	82 663	92 756	81 164
WT-DFS	1 183	1 083	965	614	609	594	35 495	35 212	5 501

considerably faster than scanning the whole text for Kernel and CC. We also observe the WT-DFS is less dependent on the number of subpatterns k than SA-SCAN, since no overhead for copying and explicitly sorting SA ranges is required. Also WT-DFS profits from larger minimum gap sizes as larger parts of the text are skipped when gap constraints are violated near the root of the WT. For Dna-Hg38, small subpattern length of $m_i = 3$ generate large SA intervals which in turn decrease query performance comparable to processing the complete text.

Matching Performance for Different Subpattern Lengths. In the second experiment, we measure the impact of subpattern lengths on query time. We fix

Table 3. Space usage relative to text size at query time of the different indexes for three data sets of size 2 GiB, different subpattern lengths $m_i \in 3, 5, 7$ and varying number of subpatterns $k \in 2, 4, 8, 16, 32$

Method	Kernel-2G			CC-2G			Dna-Hg38-2G		
	3	5	7	3	5	7	3	5	7
$k = 2$									
RGXP	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
QGRAM-RGXP	7.93	7.93	7.93	7.49	7.49	7.49	7.94	7.94	7.94
SA-SCAN	5.01	5.00	5.00	5.01	5.00	5.00	5.20	4.90	4.88
WT-DFS	5.50	5.50	5.50	5.50	5.50	5.50	5.41	5.38	5.38
$k = 4$									
RGXP	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
QGRAM-RGXP	7.93	7.93	7.93	7.49	7.49	7.49	7.94	7.94	7.94
SA-SCAN	5.01	5.01	5.00	5.02	5.01	5.00	5.89	4.94	4.89
WT-DFS	5.50	5.50	5.50	5.50	5.50	5.50	5.38	5.38	5.38
$k = 8$									
RGXP	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
QGRAM-RGXP	7.93	7.93	7.93	7.49	7.49	7.49	7.94	7.94	7.94
SA-SCAN	5.06	5.02	5.01	5.06	5.02	5.01	6.57	5.03	4.91
WT-DFS	5.50	5.50	5.50	5.50	5.50	5.50	5.38	5.38	5.38
$k = 16$									
RGXP	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
QGRAM-RGXP	7.93	7.93	7.93	7.49	7.49	7.49	7.94	7.94	7.94
SA-SCAN	5.22	5.09	5.02	5.18	5.09	5.03	7.86	5.25	4.94
WT-DFS	5.50	5.50	5.50	5.50	5.50	5.50	5.38	5.38	5.38
$k = 32$									
RGXP	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
QGRAM-RGXP	7.93	7.93	7.93	7.49	7.49	7.49	7.94	7.94	7.94
SA-SCAN	5.32	5.09	5.04	5.31	5.11	5.03	10.20	5.54	5.00
WT-DFS	5.50	5.50	5.50	5.50	5.50	5.50	5.38	5.38	5.38

the gap constraint to $\langle 100, 110 \rangle$ and the data sets size to 2 GiB. Table 2 shows the results. Larger subpattern length result in smaller SA ranges. Consequently, query time performance of SA-SCAN and WT-DFS improves. As expected RGXP performance does not change significantly, as the complete text is scanned irrespectively of the subpattern length.

Matching Performance for Different Text Sizes. In this experiment we explore the dependence of query time on text size. The results are depicted in Fig. 4. The boxplot summarizes query time for all ks and all gap constraints for a

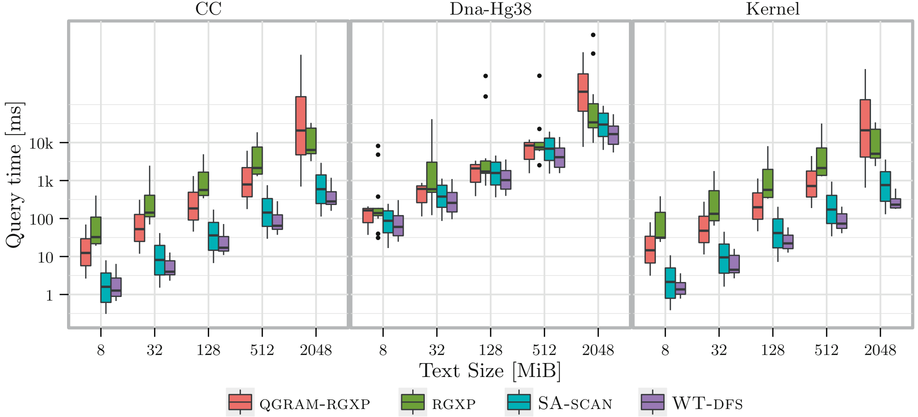

Fig. 4. Average query time dependent on input size for subpattern length $m_i = 3$.

fixed subpattern length $m = 3$. As expected, the performance of RGXP increases linearly with the text size for all datasets. The indexed solutions QGRAM-RGXP SA-SCAN and WT-DFS also show a linear increase with dataset size. We observe again that the SA based solutions are significantly faster than the RGXP base methods. For CC and Kernel the difference is one order of magnitude – even for small input sizes of 8 MiB. We also observe that WT-DFS is still the method of choice for the unfavorable case of a small alphabet text with small subpattern length of $m = 3$.

Space Usage at Query Time. In addition to run time performance, we evaluate the in memory space usage of the different indexes at query time. The space usage considered is the space of the underlying index structure in addition to the temporary space required to answer queries. For example, the SA-SCAN method requires additional space to sort the positions of each subinterval to perform efficient intersection. Similarly, the QGRAM-RGXP index requires additional space to store results of intersections for subpatterns larger than q. The space usage of the different index structures relative to the text size is shown in Table 3. Clearly RGXP requires only little extra space in addition to the text to store the regexp automaton. The QGRAM-RGXP requires storing the text, the compressed q-gram lists, the regexp automaton for verification, and during query time, q-gram intersection results. The SA-SCAN index requires storing the suffix array ($n \log n$ bits), which requires roughly $4n$ bytes of space for a text of size 2 GiB plus the text (n bytes) to determine the subpattern ranges in the suffix array. Additionally, SA-SCAN requires temporary space to sort subpattern ranges. Especially for frequent subpatterns, this can be substantial. Consider the Dna-Hg38 dataset for $k = 32$ and $m = 3$. Here the space usage of SA-SCAN is $9n$, which is roughly twice the size of the index structure. This implies that SA-SCAN potentially requires large amounts of additional space at query time which can be prohibitive. The WT-DFS index encodes the suffix array using a

wavelet tree. The structure requires $n \log n$ bits of space plus $o(n \log n)$ bits to efficiently support rank operations. In our setup we use an rank structure which requires 12.5% of the space of the WT bitvector. In addition, we store the text to determine the suffix array ranges via forward search. This requires another $n \log \sigma$ bits which corresponds to n bytes for CC and CC. For this reason the WT-DFS index is slightly larger than SA-SCAN. We note that the index size of WT-DFS can be reduced from $5.5n$ to $4.5n$ by not including the text explicitly. The suffix array ranges can still be computed with a logarithmic slowdown if the WT over the suffix array is augmented with select structures. The select structure enables access to the inverse suffix array and we can therefore simulate Ψ and LF. This allows to apply backward search which does not require explicit access to the original text.

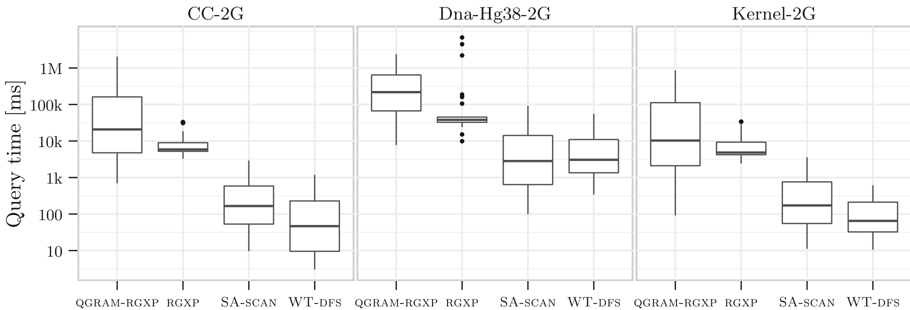

Fig. 5. Overall runtime performance of all methods for three data sets, accumulating the performance for all $m_i \in 3, 5, 7$ and C_S, C_M and C_L for text size 2 GiB.

Overall Runtime Performance. In a final experiment we explored the whole parameter space (i.e. $k \in \{2^1, \dots, 2^5\}$, $m_i \in \{3, 5, 7\}, C \in \{C_S, C_M, C_L\}$) and summarize the results in Fig. 5. Including also the large subpattern length $m_i = 5$ and $m_i = 7$ results in even bigger query time improvement compared to the RGXP based approaches: for CC and Kernel SA based method queries can be processed in about 100 ms while RGXP require 10 s on inputs of size 2 GiB. The average query time for Dna-Hg38 improves with SA based methods from 50 s to 5 s. The WT based approach significantly improves the average time for CC and Kernel and is still the method of choice for Dna-Hg38.

5 Conclusion

In this paper we have shown another virtue of the wavelet tree. Built over the suffix array its structure allows to speed up variable length gap pattern queries by combining the sorting and filtering process of suffix array based indexes. Compared to the traditional intersection process it does not require copying of data and enables skipping of list regions which can not satisfy the intersection

criteria. We have shown empirically that this process outperforms competing approaches in many scenarios.

In future work we plan to reduce the space of our index by not storing the text explicitly and using the wavelet tree augmented with a select structure to determine the intervals of the subpatterns in the suffix array.

Acknowledgement. We are grateful to Timo Bingmann for profiling our initial implementation. This work was supported under the Australian Research Council's Discovery Projects scheme (project DP140103256) and Deutsche Forschungsgemeinschaft.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
2. Baeza-Yates, R.: A fast set intersection algorithm for sorted sequences. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 400–408. Springer, Heidelberg (2004)
3. Bille, P., Gørtz, I.L.: Substring range reporting. Algorithmica **69**(2), 384–396 (2014)
4. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: Proceedings of SODA, pp. 1297–1308 (2010)
5. Bille, P., Gørtz, I.L., Vildhøj, H.W., Wind, D.K.: String matching with variable length gaps. Theor. Comput. Sci. **443**, 25–34 (2012)
6. Fredriksson, K., Grabowski, S.: Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. Inf. Retrieval **11**(4), 335–357 (2008)
7. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 326–337. Springer, Heidelberg (2014)
8. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of SODA, pp. 841–850 (2003)
9. Hulo, N., Bairoch, A., Bulliard, V., Cerutti, L., De Castro, E., Langendijk-Genevaux, P.S., Pagni, M., Sigrist, C.J.A.: The PROSITE database. Nucleic Acids Res. **34**(suppl 1), D227–D230 (2006)
10. Knuth, D.E., Morris Jr., J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
11. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Soft. Prac. Exp. **45**(1), 1–29 (2015)
12. Lewenstein, M.: Indexing with gaps. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 135–143. Springer, Heidelberg (2011)
13. Lopez, A.: Hierarchical phrase-based translation with suffix arrays. In: Proceedings of EMNLP-CoNLL, pp. 976–985 (2007)
14. Manber, U., Myers, E.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
15. Metzler, D., Croft, W.B.: A Markov random field model for term dependencies. In: Proceedings of SIGIR, pp. 472–479 (2005)
16. Mihalcea, R., Tarau, P., Figa, E.: Pagerank on semantic networks, with application to word sense disambiguation. In: Proceedings of COLING (2004)

17. Morgante, M., Policriti, A., Vitacolonna, N., Zuccolo, A.: Structured motifs search. J. Comput. Biol. **12**(8), 1065–1082 (2005)
18. Navarro, G., Raffinot, M.: Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. J. Comput. Biol. **10**(6), 903–923 (2003)
19. Rahman, M.S., Iliopoulos, C.S., Lee, I., Mohamed, M., Smyth, W.F.: Finding patterns with variable length gaps or don't cares. In: Chen, D.Z., Lee, D.T. (eds.) COCOON 2006. LNCS, vol. 4112, pp. 146–155. Springer, Heidelberg (2006)
20. Thompson, K.: Regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968)